Technical Note

1972-15

FDP Diagnostic System

P. E. Blankenship
P. G. McHugh

22 February 1972

Prepared for the Department of the Air Force
and the Advanced Research Projects Agency
under Electronic Systems Division Contract F19628-70-C-0230 by

Lincoln Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Lexington, Massachusetts

Approved for public release; distribution unlimited.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LINCOLN LABORATORY

FDP DIAGNOSTIC SYSTEM

P. E. BLANKENSHIP
P. G. McHUGH

Group 24

TECHNICAL NOTE 1972-15

22 FEBRUARY 1972

LEXINGTON MASSACHUSETTS

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored in part by the Department of the Air Force under Contract F19628-70-C-0230 and in part by the Advanced Research Projects Agency of the Department of Defense under Contract F19628-70-C-0230 (ARPA Order 1559).

ABSTRACT

A diagnostic system consisting of a series of programs to check the operation of each subset of the computer hardware is described as are the details of each diagnostic test. All fault conditions and the concomitant console display interpretation for each condition are tabulated for easy deciphering.

Accepted for the Air Force
Joseph R. Waterman, Lt. Col. USAF
Chief, Lincoln Laboratory Project Office

FDP DIAGNOSTIC SYSTEM

General Description

The Fast Digital Processor (FDP) diagnostic system is a series of programs that exercises subsets of the computer's hardware to determine a faulty section but not the faulty integrated circuit.

Conceptually, the FDP is partitioned into several separate, though unavoidably interrelated, test areas. The major test areas include: control, arithmetic elements, control memory, data memories, and Ampex core memory (with interface). The control area is divided into subsections because of its inherent complexity.

In general, each test program requires one FDP core image that is stored in the Univac 1219's Vermont Research drum system. Where possible, several test programs are squeezed into one core image to conserve drum storage space. Such images are referred to as "Subpackage A" and "Subpackage B." The images are stored according to their drum track assignments:

140	Index Memory Test
141	Unconditional Jump Test
142	Subpackage A
143	Skip/Make Diagnostic
144	Subpackage B
145	Transfer and Arithmetic Group Test
146	Multiplier and Multiplier Overflow Test
147	Program Memory/Block Transfer Test
150	Ampex Core Memory Diagnostic
151	Data Memory Test (Short Form).

An open-ended form of the Data Memory Test, used for long-term memory performance evaluation, is stored with the drum system diagnostic program.

The diagnostic system includes a drum reader routine, residing in the Univac, to read the FDP diagnostics automatically in correct sequence from the drum and bootstrap them to the FDP.

System Operation

The greatest use of the FDP diagnostic package will probably be as part of the daily system checkout routine. Each morning, subsequent to power turn-on, the package will verify the FDP's operational status. Specific data on user complaints occurring during daily operation are also acquired quickly.

Straightforward system operation proceeds as follows:

1. Place the FDP drum reader paper tape in the 1219 tape reader.
2. Set the 1219 skip switches according to the following mode options:
   a. No Switches

      Tests residing on drum tracks 140-151 are read and executed. The 1219 then faults to DEBUG.

   b. Switch 0 Up

      Tests residing on drum tracks 140-151 are read and executed in a continuous loop. No fault to DEBUG.

   c. Switch 3 Up

      Tests residing on drum tracks 140-151 are read and executed, then the 1219 drum system test is called. An open-ended data memory test is left running in the FDP. The diagnostic loader residing in the 1219 is destroyed.

3. Type ↑L to DEBUG

The "No Switch" option obviously constitutes a single pass through the test sequence. It might be used to look for the cause of a specific user complaint. The Switch 0 option could be useful when an overnight cyclic exercise of the entire machine is desired. If a failure occurs, the FDP stops. The console display indicates the nature of the failure. To interpret the display properly it is necessary to know which diagnostic detected the fault. How long the system cycled before the fault occurred will also be of interest. The drum reader, residing in the 1219, can be interrogated to yield this information. Two registers examined via normal DEBUG techniques are interpreted as follows:

COUNT1/ (Number of diagnostic detecting failure)

COUNT2/ (Number of full test cycles before failure)

The contents of COUNT1 are associated with the diagnostic programs according to the following schedule:

COUNT1	Test Detecting Fault
1	Index Memory Test
2	Unconditional Jump Test
3	Subpackage A
4	Skip/Make Test
5	Subpackage B
6	Arithmetic Element/Control Group Test
7	Multiply/Overflow Test
10	Program Memory Test

11 Ampex Core Memory Test

12 Data Memory Test

To get the approximate running time before the fault occurred, the contents of COUNT2 should be converted to decimal radix and multiplied by 77. This yields the time in seconds. For example, assume COUNT2 = 127,

$$127_8 = 87_{10}$$

$$\frac{87 \times 77}{3600} \cong 1.86 \text{ hr}$$

The continuous test loop cycle can be interrupted by raising the STOP 0 switch. When the 1219 has stopped, DEBUG can be re-entered by standard methods.

If SKIP 3 switch is up, all diagnostics will be called from the drum in proper sequence, then the 1219 drum test will be read automatically into core. This leaves an open-ended, continuous data memory check running concurrently in the FDP. If an FDP memory fault is detected, it is reported automatically to the drum system test program running in the 1219. Error messages will be typed out on-line to give details of the failure mode. The drum test overlays the FDP drum reader program when it is loaded into core. If further diagnostic manipulation is to be performed, the drum test must be interrupted (by standard means) and the FDP drum reader reloaded.

If the entire drum facility fails, a backup of the diagnostic system exists on paper tape. Each FDP image can be loaded and executed individually from the paper tape.

FDP Drum Reader

The FDP drum reader program (Fig. 1) is reasonably self-explanatory. The drum directory is initialized with the values of the tracks to be read. Thus the time counter (COUNT2) is cleared. STOP switch 0 is then checked for a manual interrupt. If none occurs, the program count (COUNT1) and the drum track pointer (TRACK) are initialized. The program count and track pointer are incremented and the track counter is checked to see if program 12 has been read. If it has not, the drum file selected by TRACK is read into Univac core. If a read parity error is detected, the file is read again. Control passes to the FDP loader, which sends the selected diagnostic to the FDP and control returns to the point shown. If the FDP has stopped due to some error, the drum reader will hang up in the FDP loader attempting to bootstrap a new file over to the FDP.

If program12 has been successfully sent to the FDP, SKIP switch 3 is checked. If up, control passes to the DRUM FDP TEST, which is the open-end data memory diagnostic as far as the FDP is concerned. If the switch is not up, SKIP switch 0 is checked. If up, the time counter is incremented and the cycle is repeated. Each cycle requires about 77_{10} seconds. Thus the total running time can be ascertained by converting the contents of COUNT2 to decimal radix and multiplying by 77/3600 hours.

This program starts at location 1000_8 in Univac 1219 core. Once loaded, the program will remain in core unless the drum test is called. Then it must be reloaded.

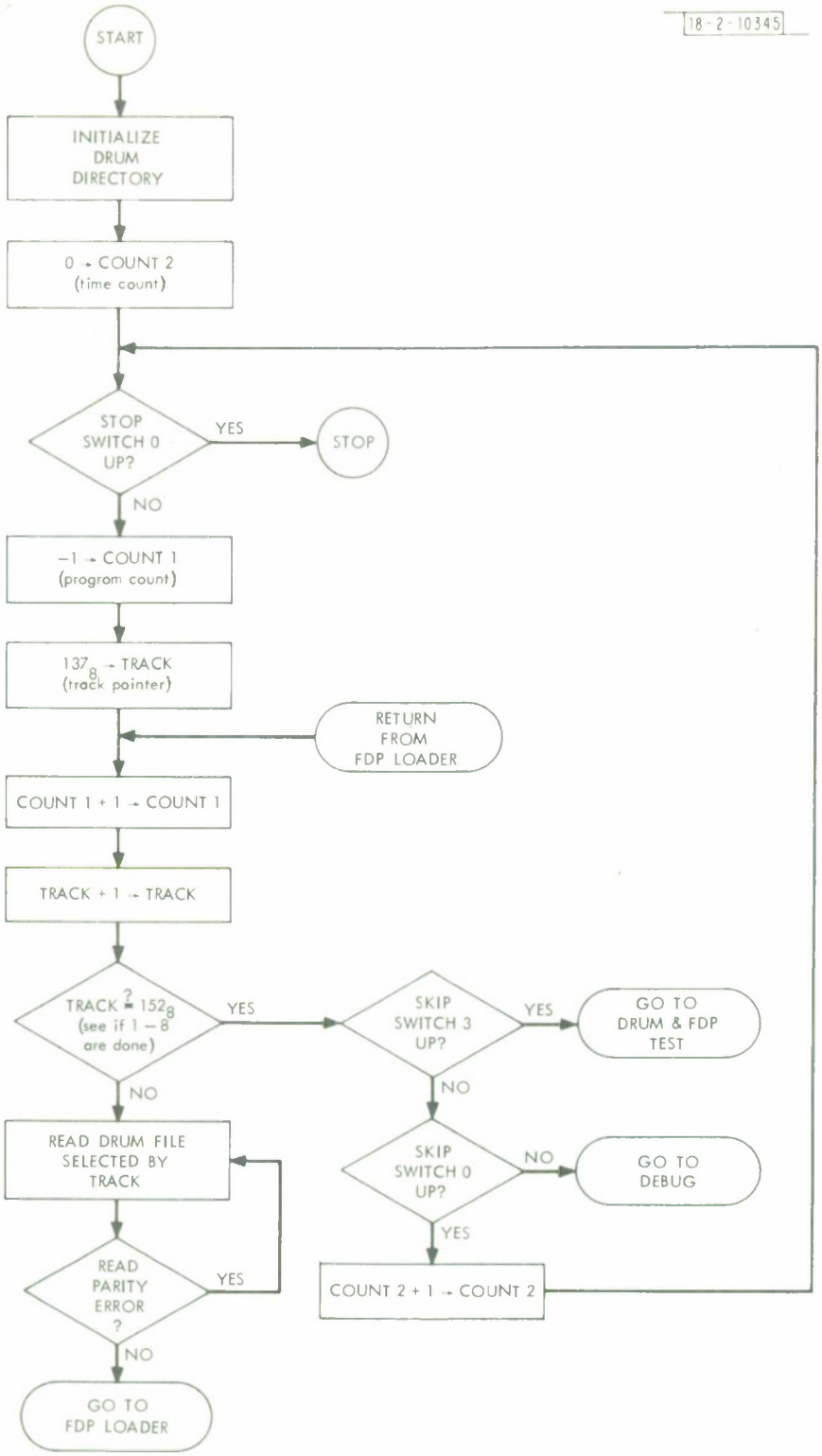


Fig. 1. FDP drum reader.

Control Diagnostics

Index Memory Test

The index memory test (Fig. 2) establishes the functionality of X_c and the associated skip commands. The test is performed by alternately clearing and setting all bits of each index memory location in succession (1 through 17_8). The contents of each location are checked via the index skip instructions. The stop conditions are:

P = 10: X(n) failed to clear

P = 13: X(n) failed to set

Examine bits 1-4 of $M_c(7)$ to establish n. Bring X(n) into lights to see faulty bits.

Unconditional Jump Test

This rather complicated exercise (Fig. 3) attempts to evaluate all hardware associated with "jump" type control transfers. The basic concept is to transfer program control to random points within the program memory, M_c. A return jump is then effected from the random location to the main body of code as follows:

1. All M_c locations, except the few containing the test program, are filled with HLT instructions. This prevents the program from "running away" if a jump occurs to an erroneous location. If so, the machine stops.

2. The program generates a random number and extracts the eight Y-field bits from it. If the Y-field does not point to an M_c location containing program code, a return jump is written into the selected location.

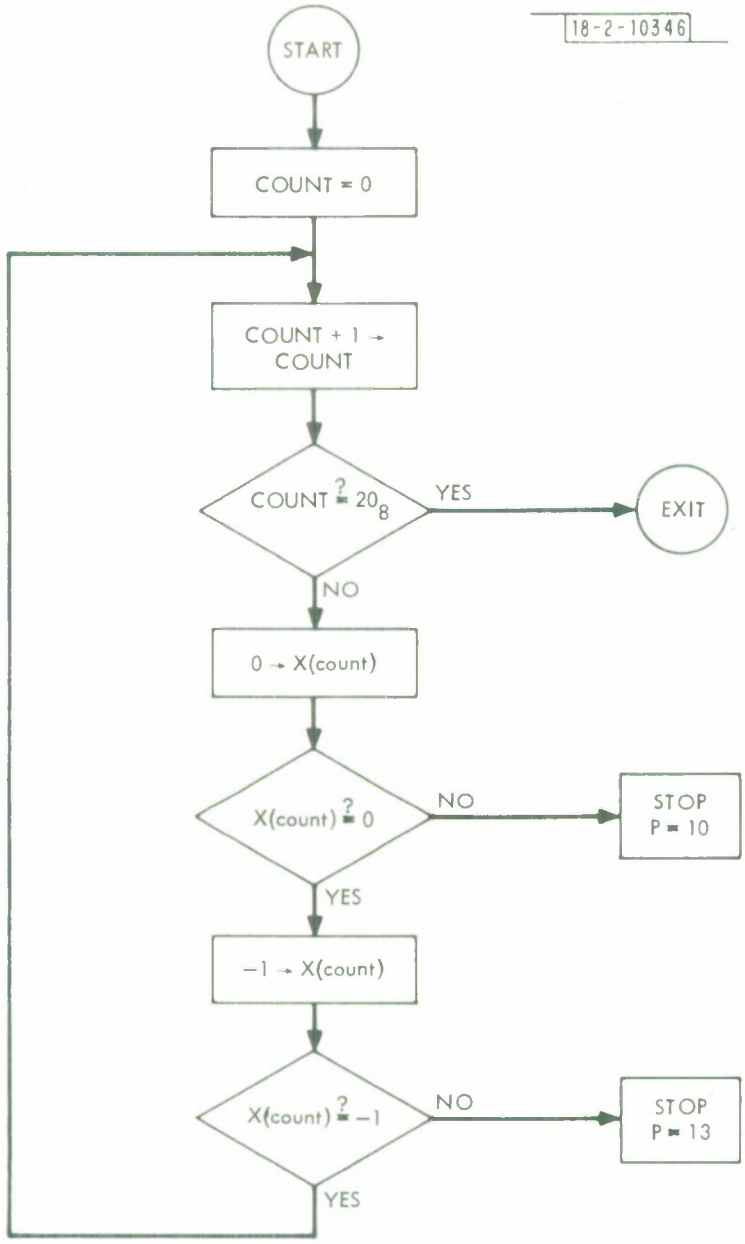


Fig. 2. Index memory test.

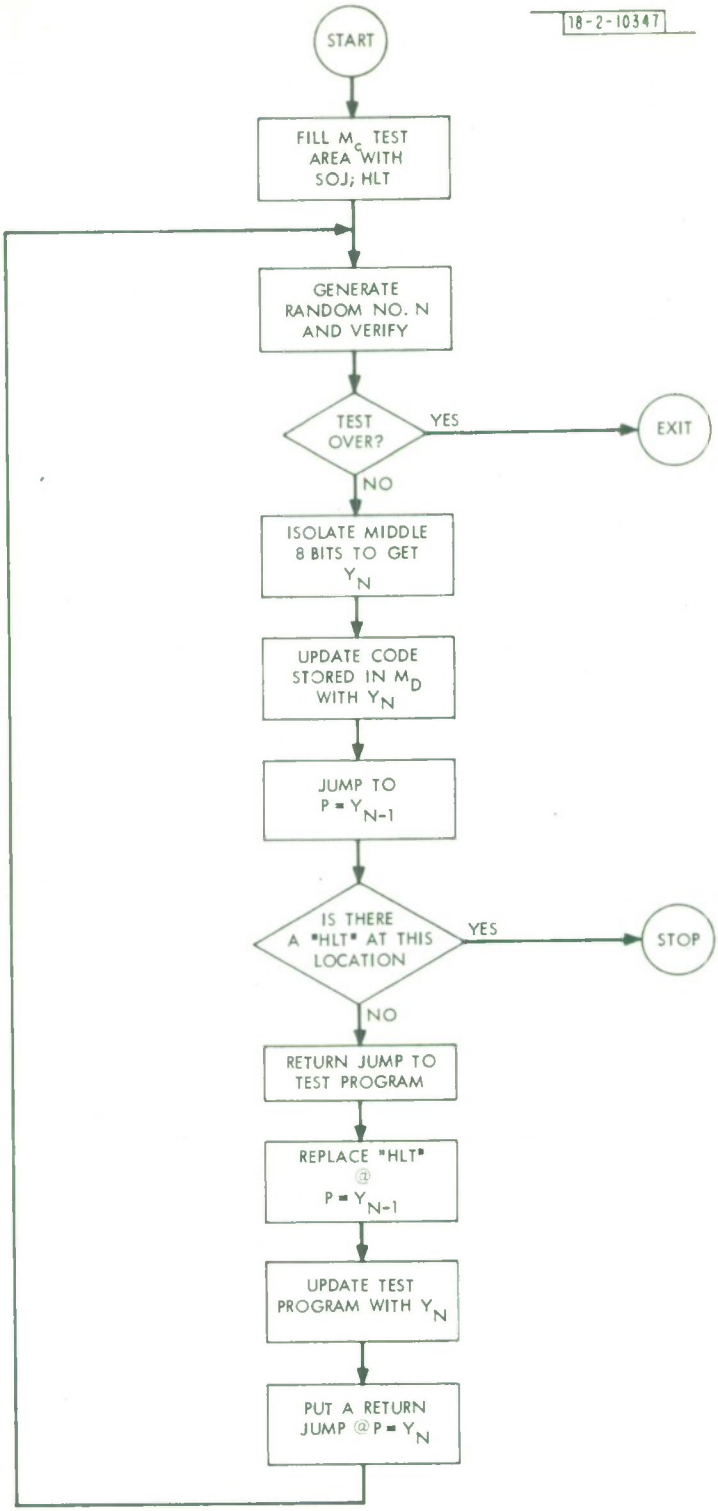


Fig. 3. Jump test.

3. The program attempts to jump to the location specified by Y. If successful, the return jump is encountered and control returns to the main program.

4. HLT is rewritten into the location selected by Y, erasing the return jump.

The cycle repeats with the generation of a new random number. If stoppage occurs, several bits of data must be gathered to infer the fault. Examine:

$M_c(31)$ (right): Y bits show where jump was to go.

$M_c(36)$ (right): Y bits show where return was written.

Check to see if return jump (XJP) is in correct location and note actual point of stoppage (P-2). Either the return jump is in the wrong place or the jump was to the wrong place.

Establish which.

Indexing Jump Test

This test (Fig. 4), the first of four comprising Subpackage A, exercises hardware associated with the JNX and JPX instructions. The procedure is to decrement (or increment) an index register with a JPX (or JNX) and to maintain a parallel counter in an AE. When the index register finally contains 0, the associated AE counter should also have reached 0.

If, for any reason, the tight indexing loop terminates early, the AE count will not have reached zero and a fault is registered. Possible causes of such difficulty might be problems with the auto-indexing adder or the associated zero detection logic. If the tight loop never falls through, the program will hang up. The trouble-shooter will notice this fact and manually interrupt to see where the hangup has occured.

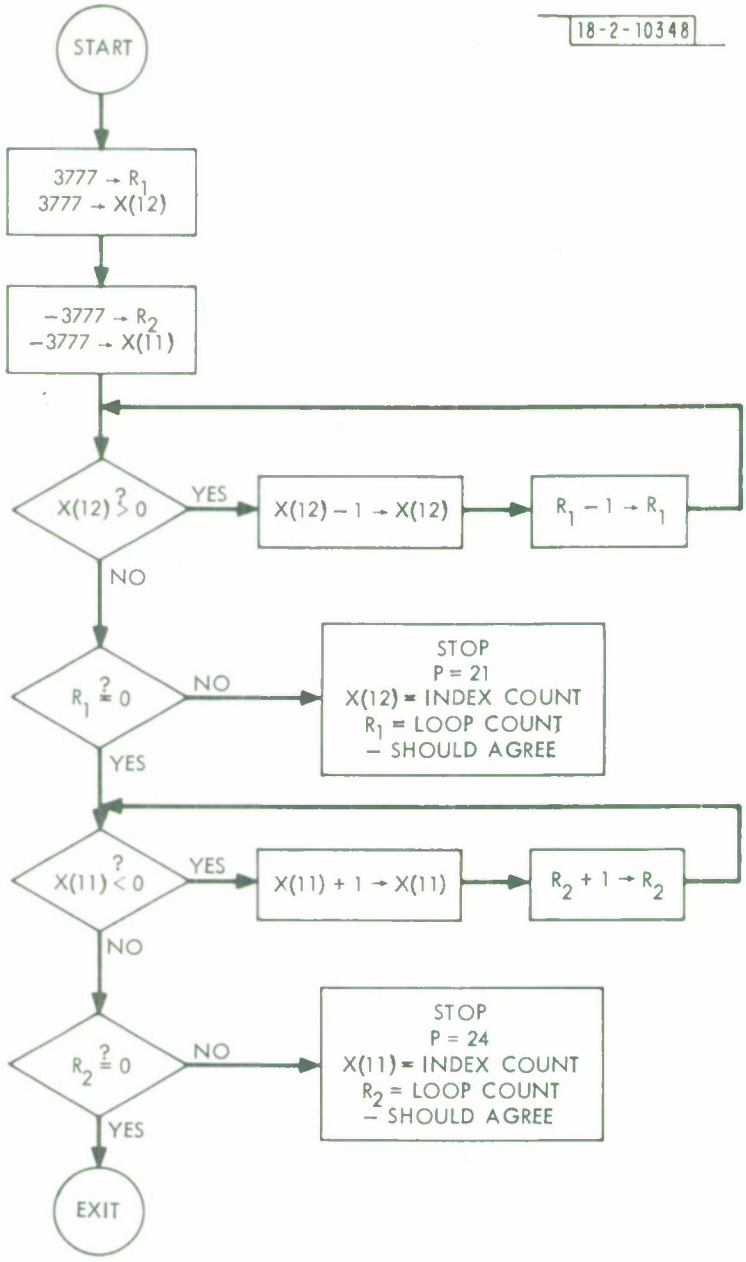


Fig. 4. Subpackage A: Auto-indexing jump test.

Error conditions are:

P = 21: JPX

X_c Lights show index register count X(12), and R_1 shows AE count. Two ought to be equal.

P = 24: JNX

X_c Lights show index register count X(11), and R_2 shows AE count. Two ought to be equal.

Arithmetic Jump Test

This test (Fig. 5), the second program of Subpackage A, embodies a rigorous test of the logic that calculates the satisfaction of an AE jump criterion as well as the hardware comprising the AE jump mechanism. The procedure involves generation of a random number in each of the four R registers, and a series of tests performed on the random number. Essentially a majority voting technique is used. Any disagreement between AEs results in a program stop. The R conditions which are tested are: $R = 0$, $R \neq 0$, $R \geq 0$, $R \leq 0$, $R > 0$ and $R < 0$. The logic tree is a bit complex. The stop conditions are:

P = 146:

Check X_c lights to see nature of failure. Refer to flow chart (Fig. 5) for exact interpretation of failure. Check R registers. All four should agree. Noting faulty AE and nature of test failed should point to jump detection logic failure.

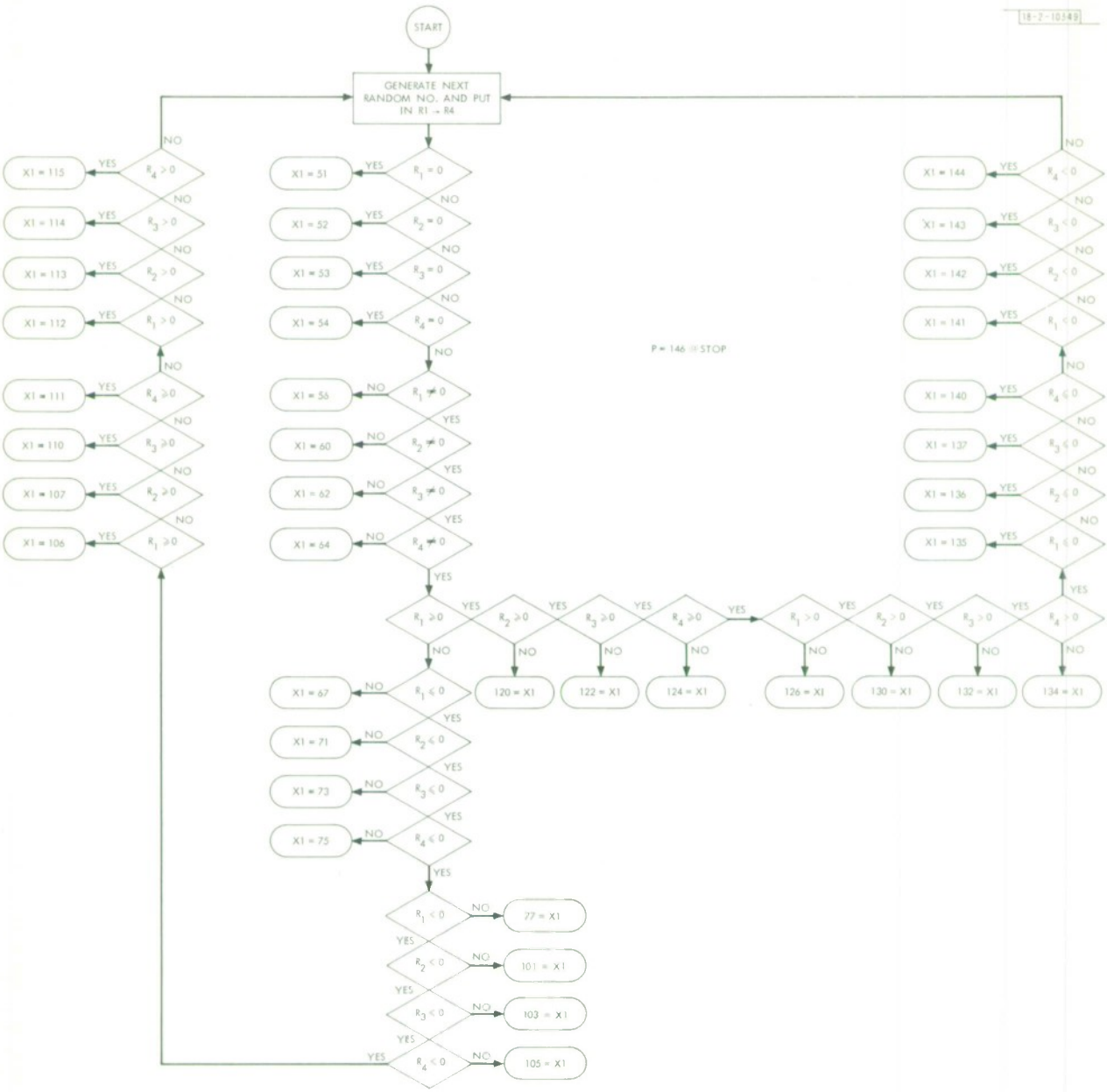


Fig. 5. Subpackage A: Arithmetic jump test.

Index Arithmetic Test

The specific intent of this test (Fig. 6), the third in Subpackage A, is to exercise the arithmetic hardware associated with the index memories. The instructions involved are YIX, YPX, YMX, and XMY.

The test works by generating random, identical, 12-bit numbers in the AEs and in the index memories. The numbers are incremented, decremented and negated/decremented in both places. At each juncture the results of the index operation are compared to those of the arithmetic element. If a disagreement is encountered, the program stops. Stop conditions are:

P = 224: FAULT

X(1) = 175

Disagreement in calculating N + 1

R_1 = N, R_2 = Difference between results, X(17) = N + 1

X(1) = 207

Disagreement in calculating N - 1

R_1 = N, R_2 = Difference, X(16) = N - 1

X(1) = 221

Disagreement in calculating -1-N

R_1 = N, R_2 = Difference, X(15) = -1-N

X(1) is shown in the X_c lights upon stoppage.

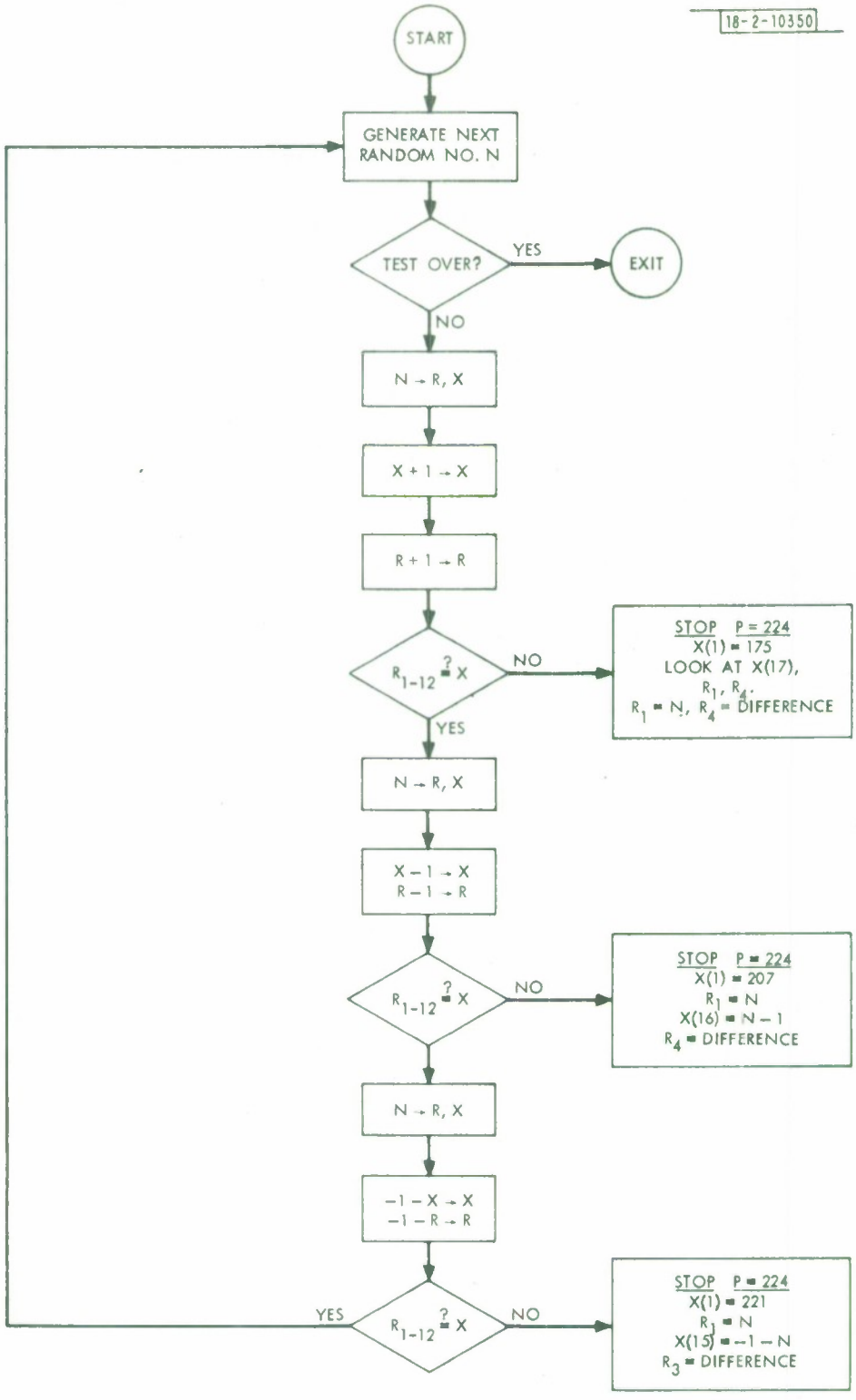


Fig. 6. Subpackage A: Index arithmetic test.

Index Skip Test

This program, (Fig. 7) the last of Subpackage A, exercises the arithmetic and comparison hardware associated with the instructions SXE, SXU, SXG and SXL. These skips base their action on calculation of the arithmetic difference between a constant (Y-field) and the contents of a selected index memory location. The difference is then evaluated according to the implicit criterion specified by the SKIP.

The test operates by generating a 12-bit random number and storing it in a prescribed index register. The number is guaranteed not to be 3777 or -4000 because these will cause the 12-bit adder to overflow. The number is then tested by a sequence of skip commands that look for logical contradictions. The various stop conditions are:

P = 303, X(1) = 255:

Claimed X = 0 when it was not.

P = 303, X(1) = 264:

Claimed X = 0 after it had decided $X \neq 0$.

P = 303, X(1) = 270.

Claimed $X < -1$ after saying $x > 0$.

P = 303, X(1) = 273:

Claimed $X < 0$ after saying $X > 0$.

P = 303, X(1) = 275:

Claimed $X > 0$ after saying $X < 0$.

P = 303, X(1) = 277:

Claimed $X > 1$ after saying $X < 0$.

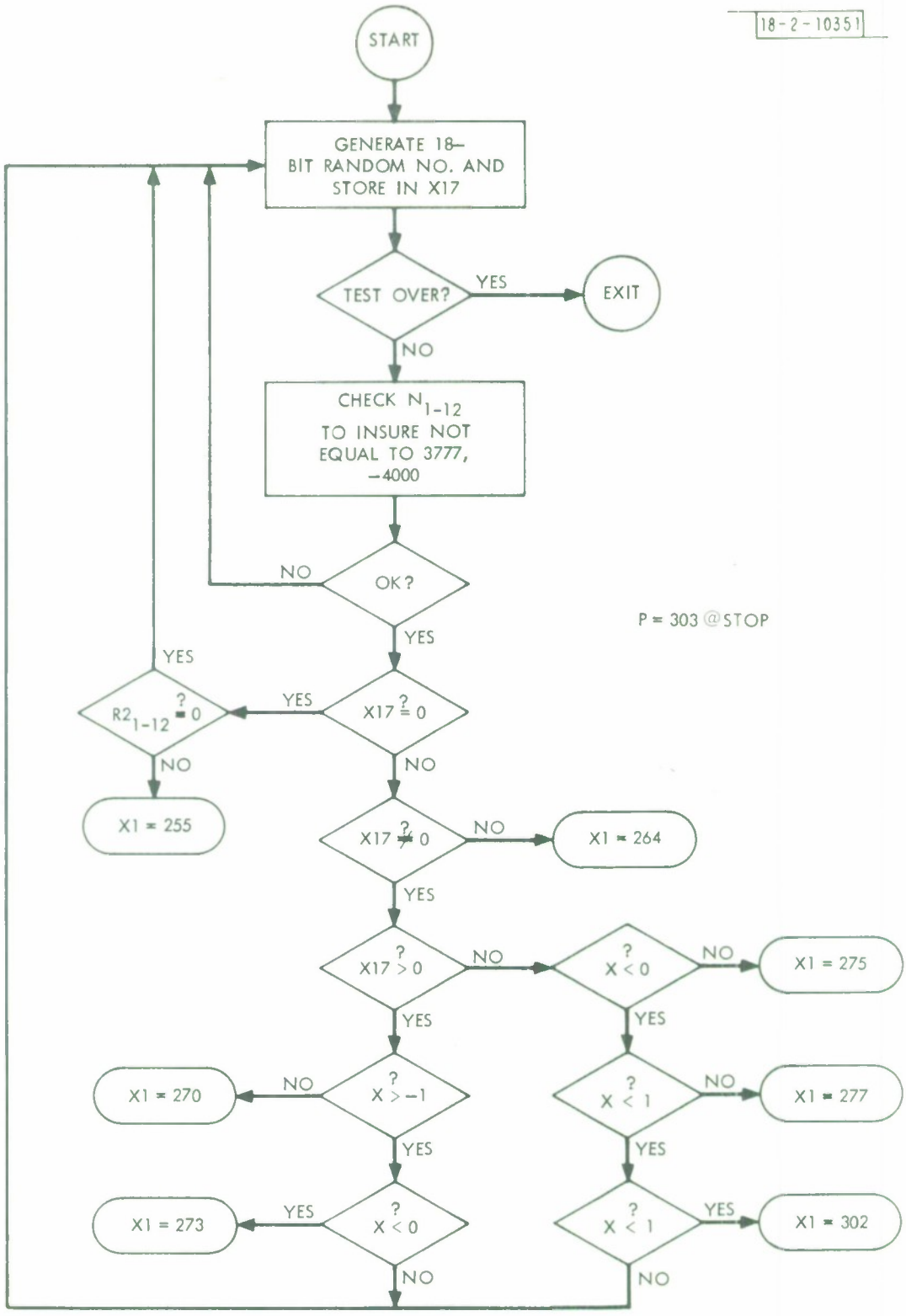


Fig. 7. Subpackage A: Index skip test.

P = 303, X(1) = 302:

Claimed X > 0 after saying X < 0.

$X_{A,\ B}$ lights display X(1).

X_C lights display N.

Skip/Make Test

This test (Fig. 8) exercises logic associated with the skip/make instruction, one which is rather unique to the FDP. A series of tests are performed in each of the 20_8 flags (0-17). The tests consist of various skip sequences based on certain prescribed flag settings. If appropriate instructions are <u>not</u> skipped, the program stops. The stop conditions are:

P = 6: Failure to unconditionally skip. Flag (N) = 0

P = 10: Failure to skip first of next two instructions.

Flag (N) = 0

P = 11: Failure to skip second of next two instructions.

Flag (N) = 0

P = 13: Failure to skip first of next three instructions.

Flag (N) = 1

P = 14: Failure to skip second of next three instructions.

Flag (N) = 1

P = 15: Failure to skip third of next three instructions.

Flag (N) = 1

P = 17: Failure to skip first of next four instructions.

Flag (N) = 0

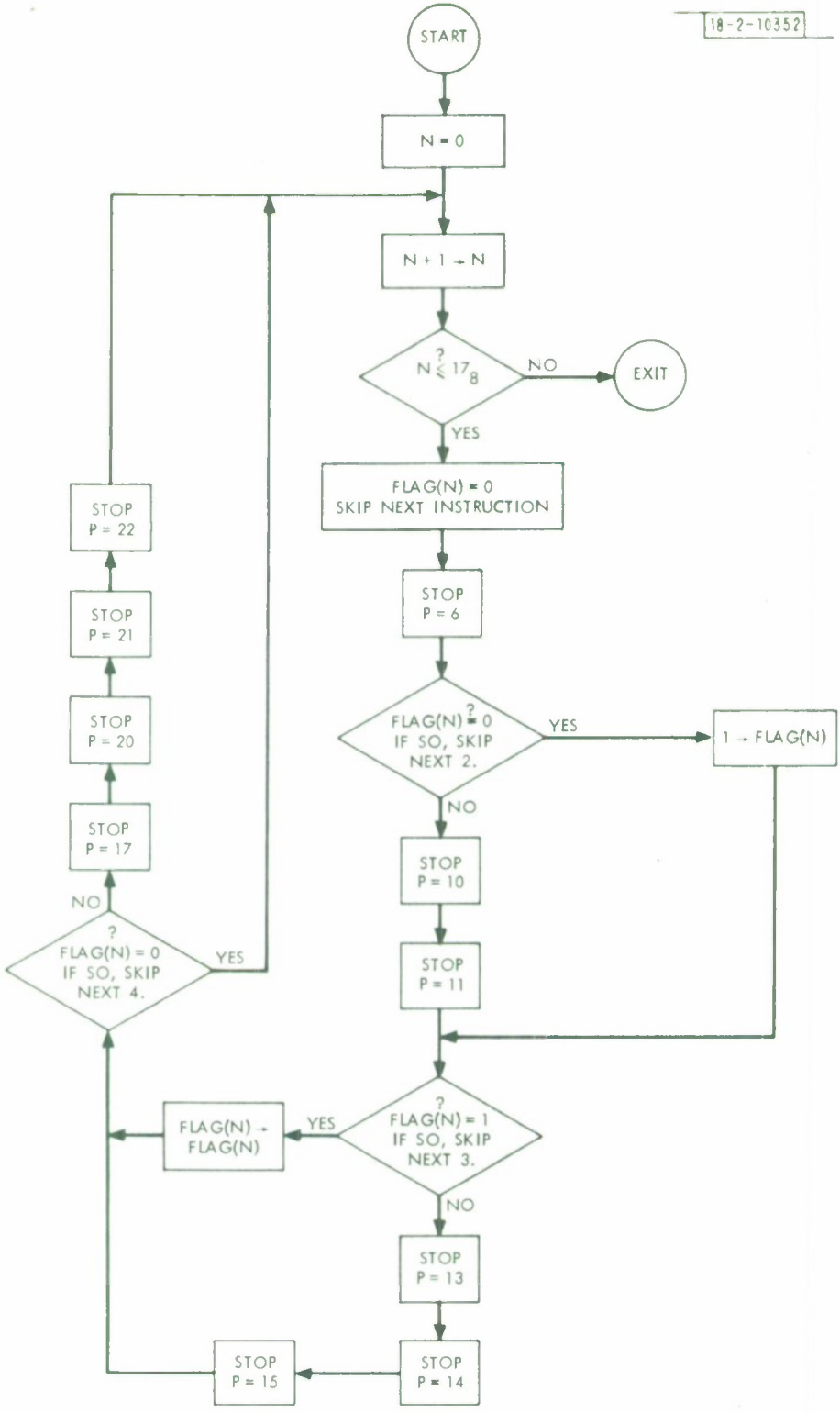


Fig. 8. Skip/make test.

P = 20: Failure to skip second of next four instructions.
Flag (N) = 0

P = 21: Failure to skip third of next four instructions.
Flag (N) = 0

P = 22: Failure to skip fourth of next four instructions.
Flag (N) = 0

To establish (N), look at bits 1-4 of $M_c(14)$, right half. If the failure is on a first instruction, the flag probably is bad. If the failure is on a subsequent instruction, the problem is probably in the SKM delay line.

F Group Test

The F Group test (Fig. 9), the first of Subpackage B, exercises the hardware and data paths associated with the F arithmetic element (sometimes called the XAU) (Fig. 9). It consists of three general subtests:

1. Exercise of the data paths between F and the index memories (XIF, FIX).

2. Check of the F shift capability (SFL, SFR).

3. General arithmetic check to examine the F adder (FPX, FMX).

The first test requires that all paths between F and the index memory be totally functional. If not, the program stops.

The next test requires that the F register be capable of shifting information 18 places to the left or right, without loss of any bits. If any bits are dropped or picked up, stoppage occurs.

In test three, 12-bit random numbers are generated and entered into F and X. The numbers are added and subtracted, the results being checked by a parallel operation occurring in the AEs. The basic concept here is to check the F adder

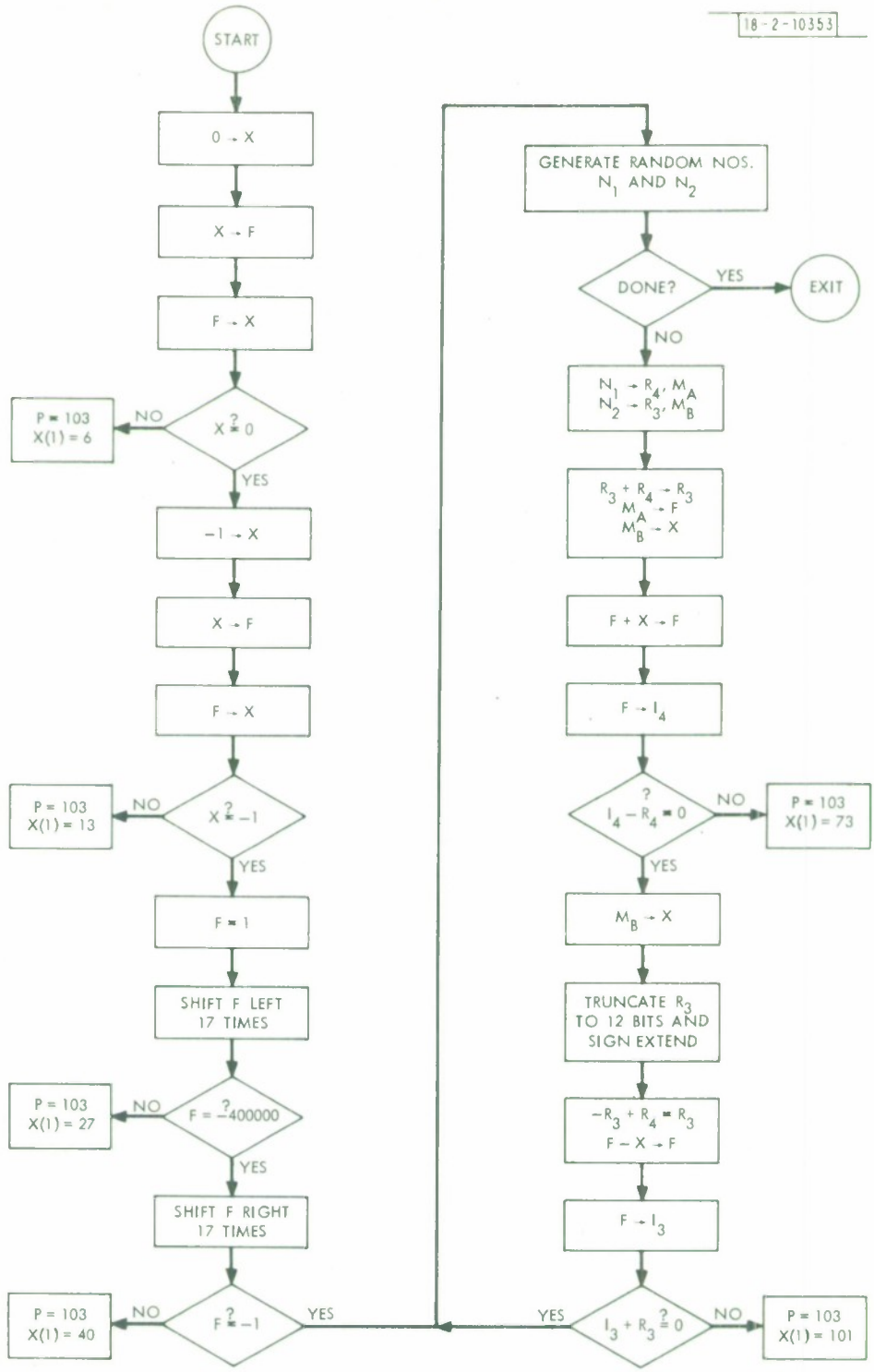


Fig. 9. Subpackage B: F group test.

complex against the adder complexes associated with the AEs. Discrepancies between the XAU and the AEs result in program stoppage. The fault conditions are:

P = 103: Failure in F group test

X(1) = 6: F to X path cannot transmit 0s.

X(1) = 13: F to X path cannot transmit 1s.

In either case check F lights to see faulty bit(s).

X(1) = 27: F dropped (or picked up) a bit in shifting to the left.

X(1) = 40: F dropped a bit in shifting to the right.

X(1) = 73: F adder failed to calculated $N_1 + N_2$. F contains $N_1 + N_2$, R_4 contains difference, R_1 and R_2 contain N_1 and N_2.

X(1) = 103: F adder failed to compute N_1 and N_2. F contains $N_1 \cdot N_2$, R_4 contains difference, R_1 and R_2 contain N_1 and N_2.

X(1) is displayed in X_c lights.

Bit-Reversed Add Test

This program, the second of Subpackage B, checks the unique FDP feature, BRA (Fig. 10). The check is effected by comparing the results of a bit-reversed add to those of a simulated bit-reversed add carried out in the AEs. The procedure is as follows:

Two random numbers are generated in AE_3 and AE_4, respectively. The numbers are bit-reverse added in the F complex, the sum residing in F. The adding process is simulated in AEs 2 and 3 using the same basic alogorithm that

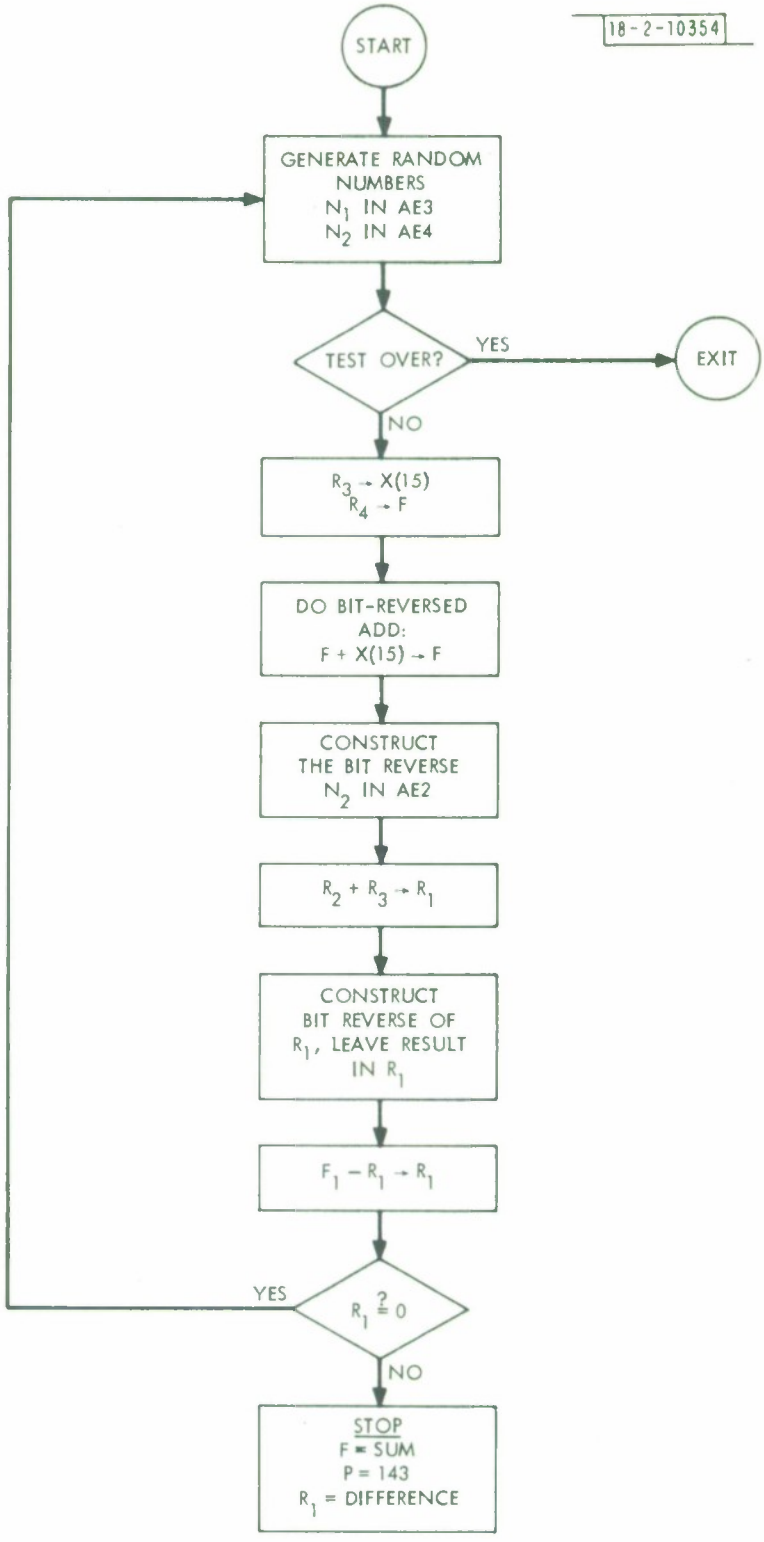


Fig. 10. Subpackage B: BRA test.

the F complex does, namely:

$$SUM = BRV\left[N_1 + BRV\ (N_2)\right] \quad .$$

It is seen that the bit-reverse of N_2 is constructed and then added to N_1. The sum is then bit reversed to obtain the final result. This result is compared (up to 12 bits) with F in R_1 (F only performs a 12-bit BRA). Discrepancies result in program stoppage. Actual construction of the bit reverse of a number proceeds as it would in any standard computer. The stop conditions are:

P = 143: Failure in bit reversed add.

F = Sum

$R_3 = N_1$

$R_4 = N_2$

R_1 = Difference.

Memory Reference Test

The memory reference test (Fig. 11), last of subpackage B, tests data paths and hardware associated with the memory transfer instruction class. It consists of four subsections:

1. Data path check
2. Inhibit capability of X_A, $X_B = 1$
3. Operability of the base address memory M_D.
4. Functionality of the address formation adder for M_A and M_B.

The first test requires that all paths between AEs and data memory, and F and data memory be capable of transmitting data accurately. This is tested by alternately setting and clearing all bits. If any bits are not functional, stoppage occurs.

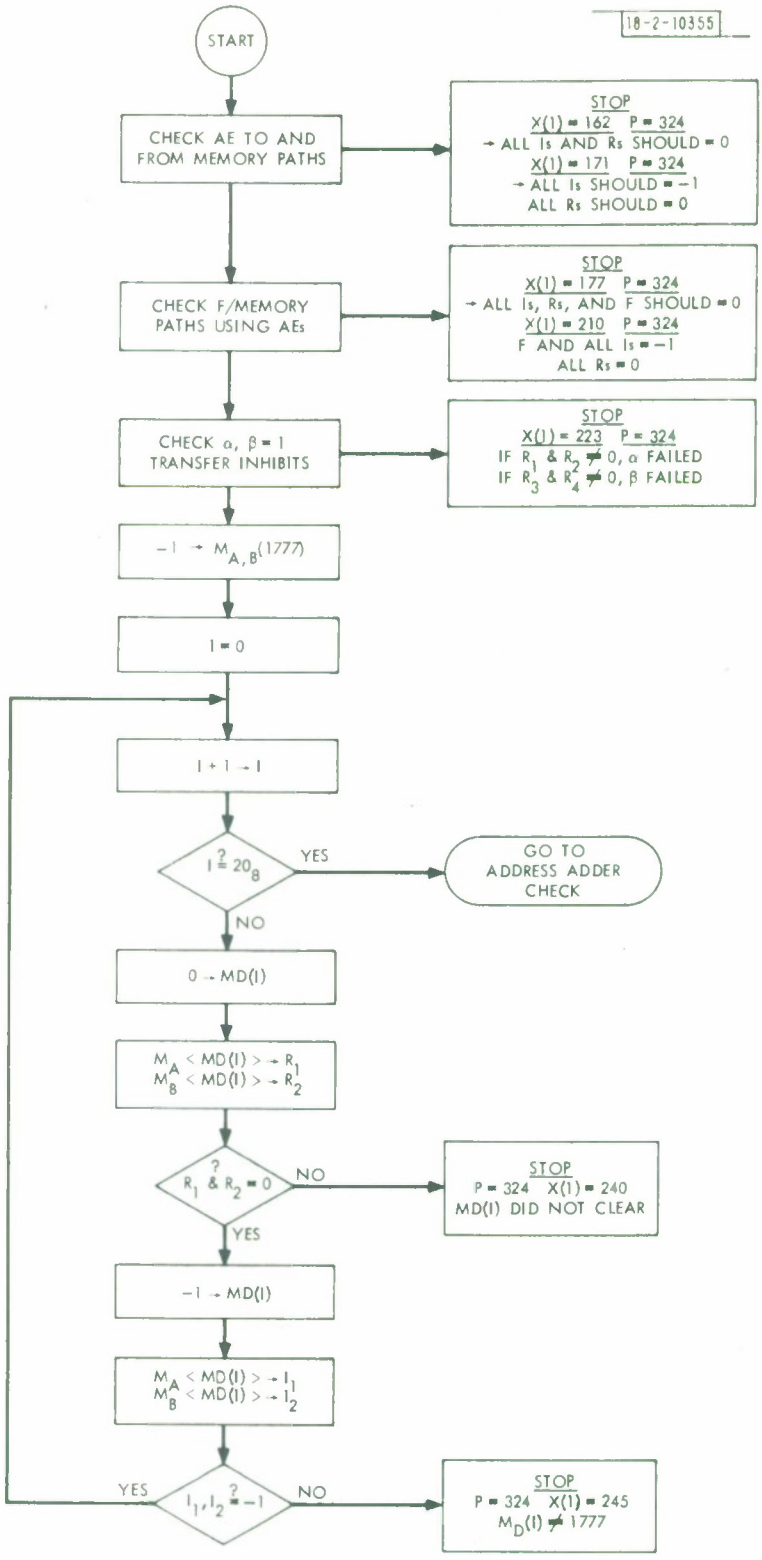


Fig. 11a. Subpackage B: Memory reference test.

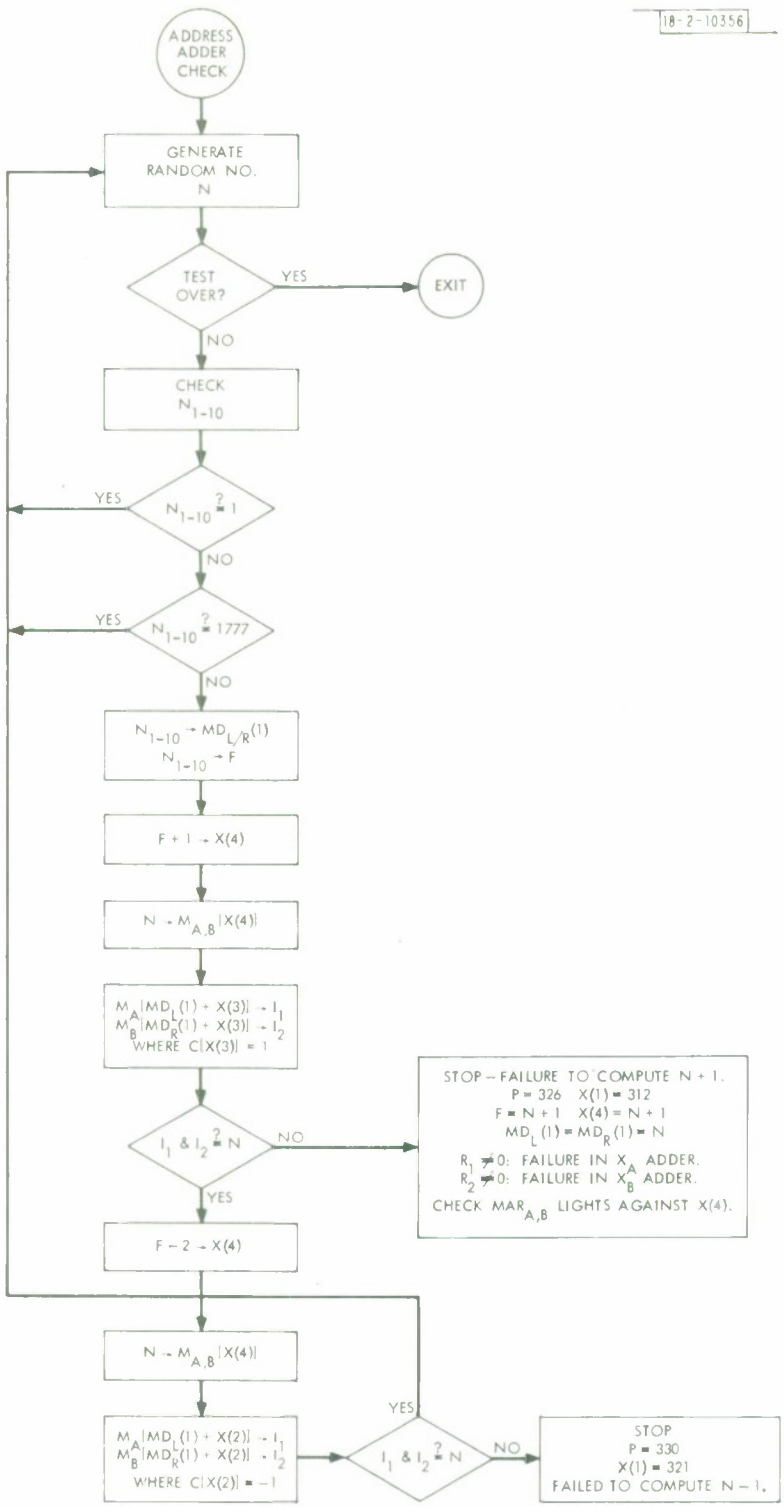


Fig. 11b. Subpackage B: Memory reference test continued.

The second test requires that the inhibiting property of a 1 in the X_A or X_B address fields of a left half instruction (α, β = 1) be functional. The test is effected by attempting to transfer two words to F. One is -1, the other all 0s. If the inhibit is not working, a non-zero result will be entered in F. If this result is detected, program stoppage occurs.

The third test requires that all M_D locations (1 - 17_8, since 0 is excluded) be fully operational. This is done by alternately clearing and setting each M_D location in succession. The correctness of a given M_D location (it must be 0 or 1777) is inferred by using the M_D location to access known data in M_A and M_B. Clearly, if M_D = 0, then 0 should be retrieved from the data memory. -1 was stored in $M_{A,B}$ (1777) and, therefore, this entity should be reaccessed when M_D = 1777. If the wrong data is detected at any given time, it is assumed that a particular M_D location failed, and program stoppage occurs.

The logic behind the address adder test assumes that the correctness of a given address is inferred by the data pointed to by the address. Presumably the data is known. The test proceeds as follows:

A random number (N) is generated and stored in X. The address (N + 1) is calculated using the F complex, and a known piece of data is stored in M_A and M_B at address (N + 1). Then the address is recalculated using the index register containing N and an M_D register containing 1. The adder must then compute the quantity (N + 1) and reaccess the known, stored piece of data. If the data fetched matches what was stored, the adder is presumed to be correct. The process is next repeated using

(N - 1). Thus the adder must be capable of computing N - 1 correctly. Any incorrect data fetches result in program stoppage. The stop conditions are:

P = 324, X(1) = 162: Failure of MEM/AE paths to transmit 0s. All Is and Rs should be 0.

Non-zero Is pinpoint bad paths.

P = 324, X(1) = 171: Failure of MEM/AE paths to transmit 1s.

All Is = -1, all Rs = 0. Any Is $\neq$ -1 show trouble.

P = 324, X(1) = 177: Failure of MEM/F paths to transmit 0s. All Is, Rs and F should be 0.

P = 324, X(1) = 210: Failure of MEM/F path to transmit 1s. All Is, F = -1; all Rs = 0.

P = 324, X(1) = 223: Failure of M_A or M_B transfer inhibit.

If R_1 and $R_2 \neq 0$, α failed.

If R_3 and $R_4 \neq 0$, β failed.

P = 324, X(1) = 240: M_D(I) location did not clear to all 0s.

P = 324, X(1) = 245: M_D(I) location did not set to all 1s.

(I) can be ascertained by examining the M_D field of M_C(234).

P = 326: Failure of address adder to computer N + 1.

P = 330: Failure of address adder to compute N - 1.

F = N + 1

M_D(1) = N

X_A, X_B lights ought to agree with F.

$R_1 \neq 0$: Failure of X_A adder.

$R_2 \neq 0$: Failure of X_B adder.

Arithmetic Element Diagnostics

The following 2 programs were designed to test the 4 arithmetic elements.

Transfer and Arithmetic Group Test

This program determines whether or not the transfer and arithmetic classes of instructions are working properly. If an instruction fails, the program indicates the failure and the logic component causing the failure via the machine's console.

The program is developed around the general philosophy that if identical operations on identical data are performed in two or more arithmetic elements, the final result in the AEs should be identical. In addition, if each operation is performed a number of times with random data, it is highly probable that faulty hardware will be detected.

For example, when the "complement R" (COR) instruction is tested (Fig. 14), an 18-bit random number, N_1, is placed in I^1, I^2, I^3 and I^4. N_1 is transferred from I to R, and then the contents of R are complemented. All the Rs should contain the same number. This is checked by performing a "subtract R forward" (SRF) instruction in each AE, which should, if there have been no errors, produce 0s in the four R registers. The program checks to see if this is actually true, and if some of the overflow flip-flops have been set to 1 as this would also indicate the instruction had failed. The COR test is performed 2^{18}-1 times using a different random number each time. If all 12^{18}-1 tests are done correctly, the COR hardware is assumed to be working correctly.

The diagnostic program (Fig. 12) begins by clearing Flag 3 and setting the F register equal to 1. Two 18-bit random numbers, N_1 and N_2, are then generated. (Flag 3 is used only when the program is testing arithmetic instructions.) If Flag 3 is a 0, the arithmetic instruction that is executed comes from the left half of the instruction register; if a 1, the arithmetic instruction comes from the right half of the instruction register. The contents of the F register indicate which instruction caused a failure. For example, if the program stops and F=5, the "change the sign of R" (CSR) instruction, which was taken from the left half of the instruction register, caused a failure. F values and their meaning are listed in Table III.

Right-half transfer instructions are tested first. The actual test logic is given in Fig. 13. If the test fails, the machine will stop with F=1. There are two causes for failure: (1) one or more R registers are non-zero, (2) some but not all the overflow flip-flops are 1s. If the R registers are non-zero the program memory address register, P, will equal 217_8. If all the R registers are 0, but there is an overflow failure, P will equal one of six numbers-- the exact number is specified by the overflow conditions (Table II).

The most common failure will be non-zero Rs. If $R^n \neq 0$, there is either a faulty component within AE^n, or in the transfer paths connecting AE^n to AE^{n-1} or to AE^{n+1}. When the failure occurs, the random numbers that were transferred between registers should be in the I registers: N_1 in I^1 and I^3, N_2 in I^2 and I^4. In any case, N_1 is always stored in M_A <100> and N_2 is stored in M_B < 100 >.

A failure can be diagnosed by stepping the machine through the transfer routine using the same operands that caused the failure and comparing the state of registers in the suspect AE^n with those in AE^{n+2} which is operating on the same

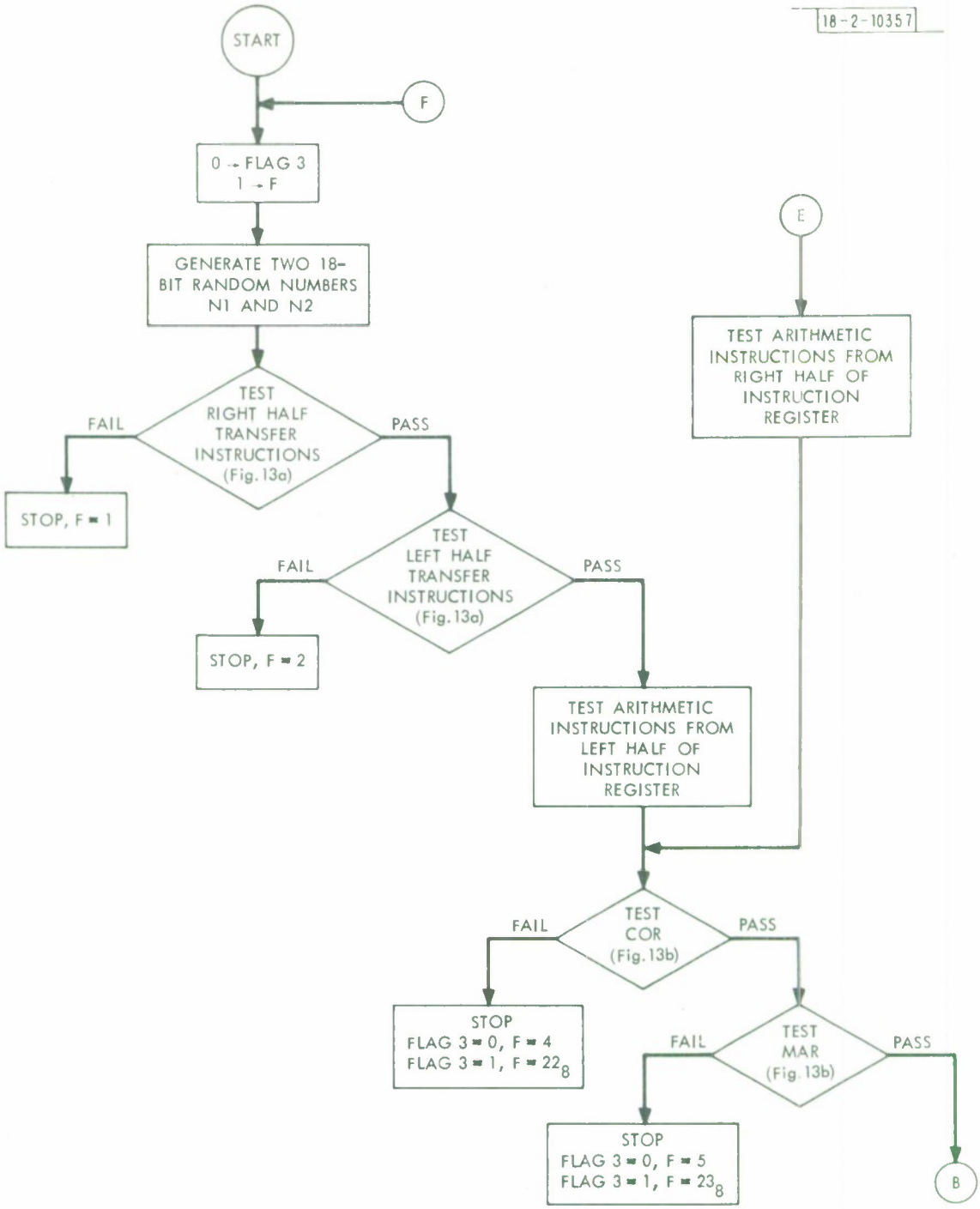


Fig. 12a. Transfer and arithmetic instruction flow diagram.

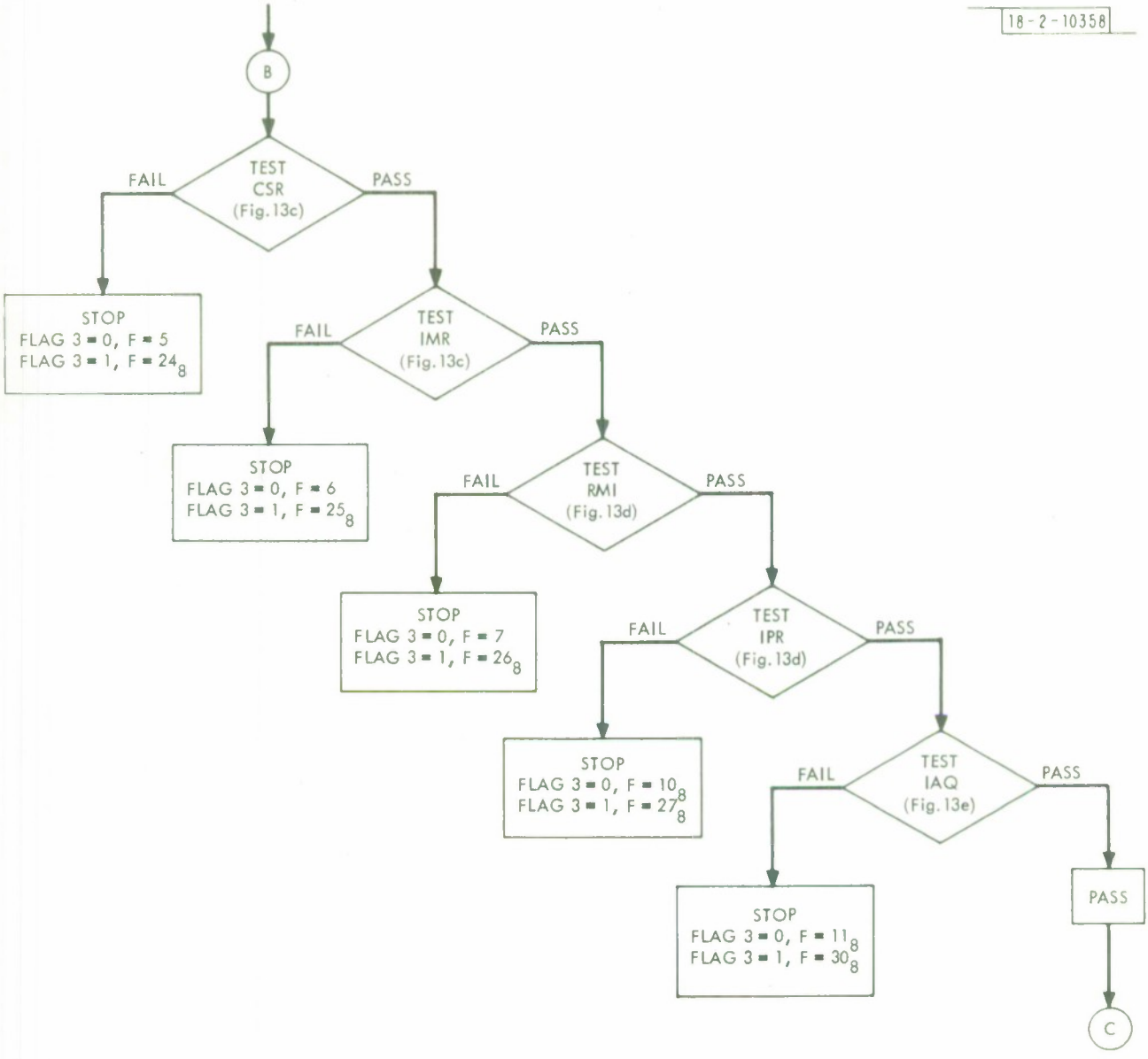


Fig. 12b. Transfer and arithmetic instruction flow diagram continued.

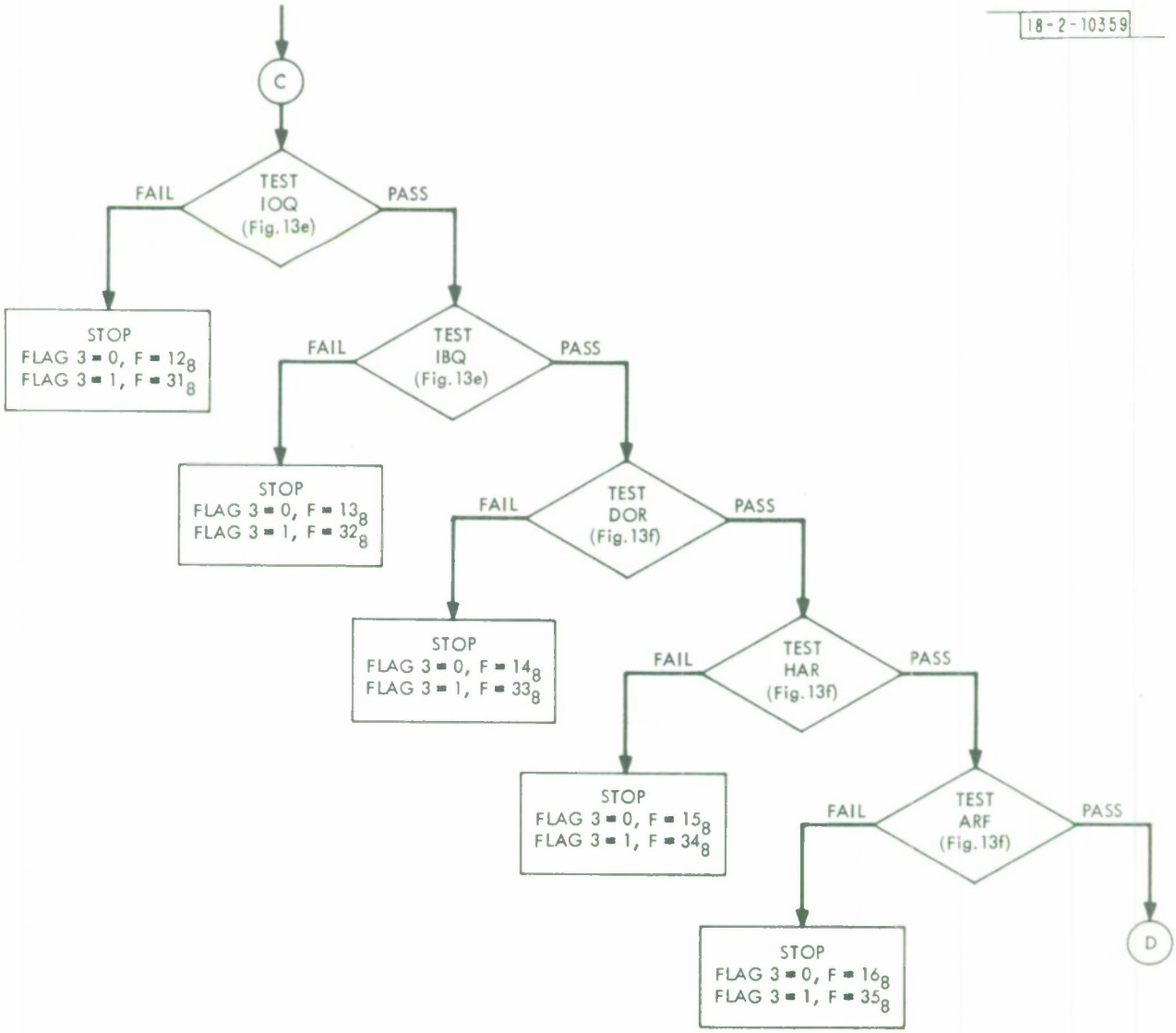


Fig. 12c. Transfer and arithmetic instruction flow diagram continued.

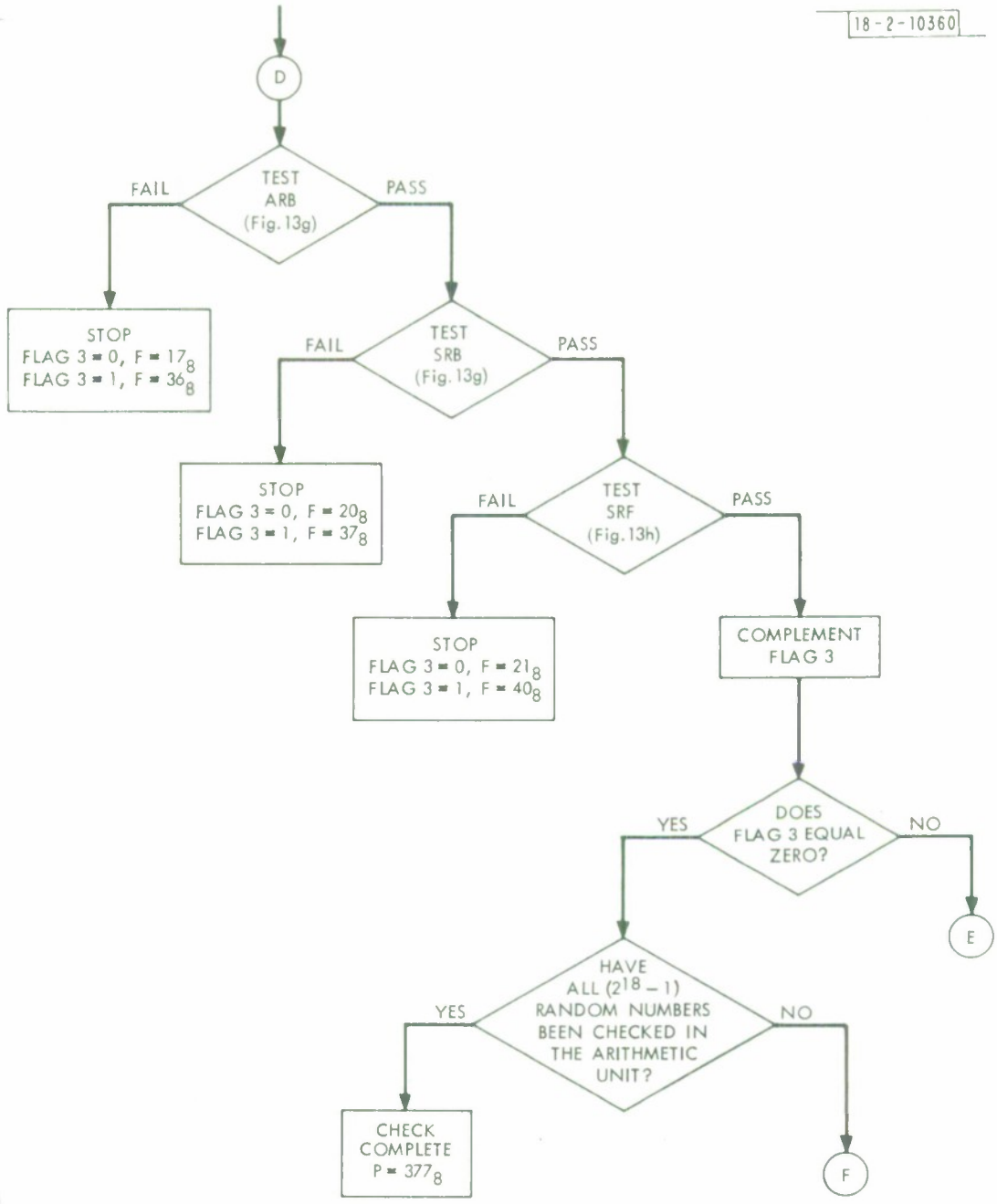


Fig. 12d. Transfer and arithmetic instruction flow diagram continued.

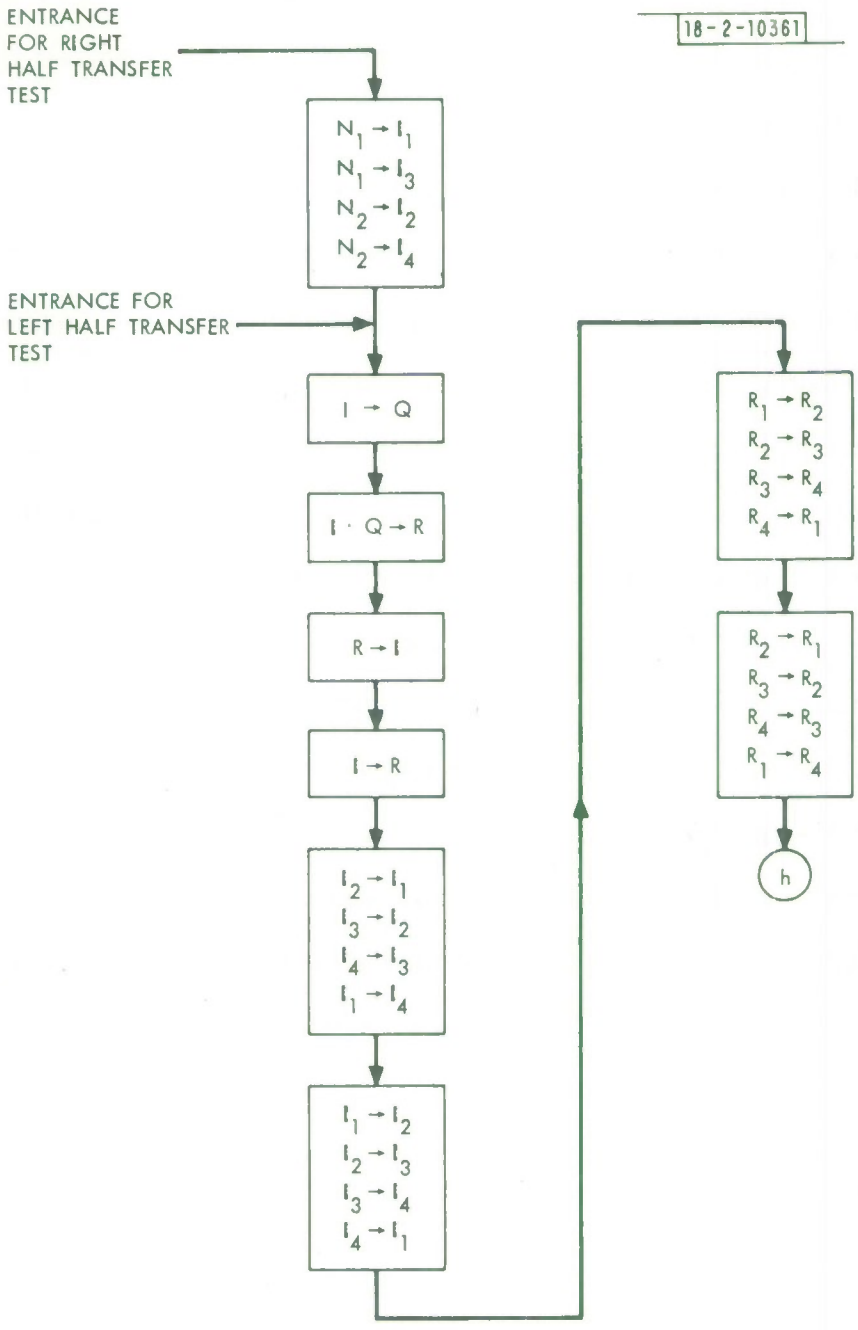


Fig. 13a. Transfer instruction test logic, left or right half.

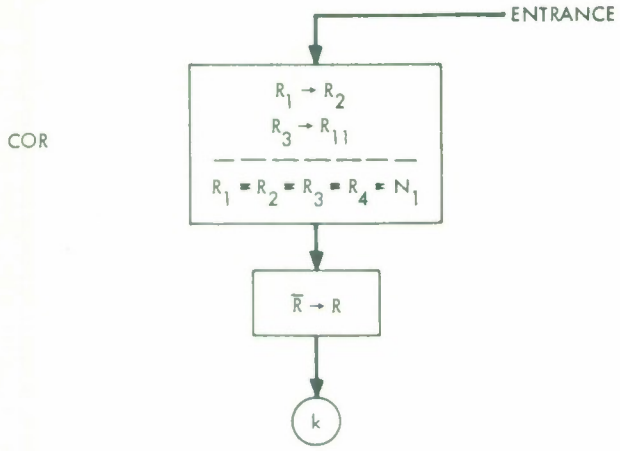
random number. This is done easily by master clearing the machine, then starting the program at $P=15_8$, and stepping the program through the right-half transfer routine which ends at $P = 25_8$.

If the machine has done all the right-half transfer instructions properly with a set of random numbers, then the transfer instructions from the left half of the instruction register are tested using the same random numbers. The only difference between a left-half instruction failure and a right-half instruction failure is that F equals 2 instead of 1. Failures are pinpointed by restarting the program at $P=15_8$, running the program until $P=27_8$, and then stepping through the right-half transfer instruction routine which begins at $P=27_8$ and ends at $P=36_8$.

Left-half arithmetic instructions are tested next, and then right-half arithmetic instructions are tested. The detailed logic for arithmetic instruction tests is given in Figs. 13b-i. F register contents for when an arithmetic instruction fails are given in Table III (Appendix).

When an arithmetic instruction operates properly, it produces 0s in all four R registers and either 1s or 0s in the overflow flip-flops. If a failure occurs in AE^n and the machine stops because of non-zero R registers, $P=217_8$, then both R^n and R^{n+1} will be non-zero. Overflow failures are indicated by specific P addresses listed in Table II (Appendix). Instructions that can and cannot have legal overflows are tabulated in Table III.

Arithmetic instruction failures are diagnosed using the method described for transfer instructions. First, master clear; then, restart program at $P=15_8$, next, run program until it reaches the starting address of the questionable routine, and then, step the machine through the routine. The starting and ending addresses for



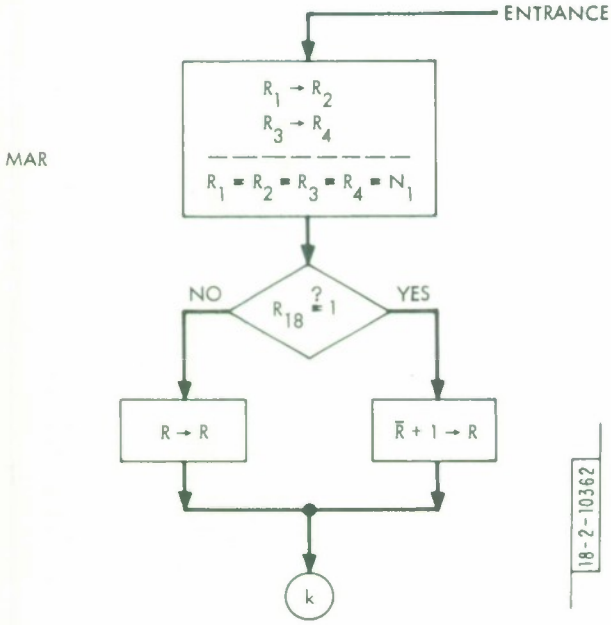
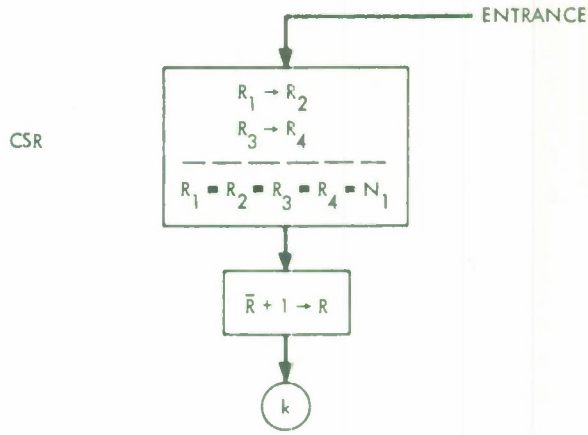


Fig. 13b. COR and MAR logic.



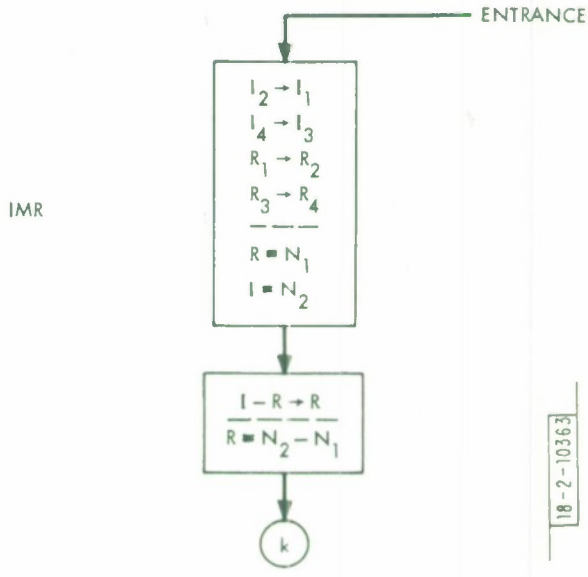


Fig. 13c. CSR and IMR logic.

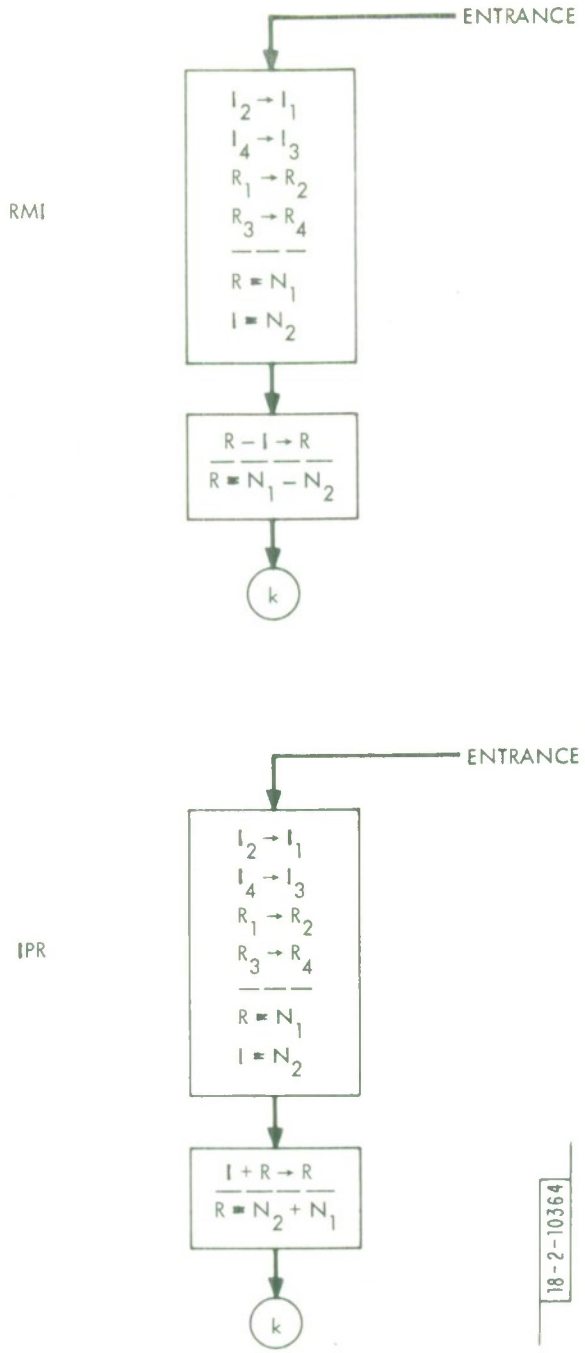


Fig. 13d. RMI and IPR logic.

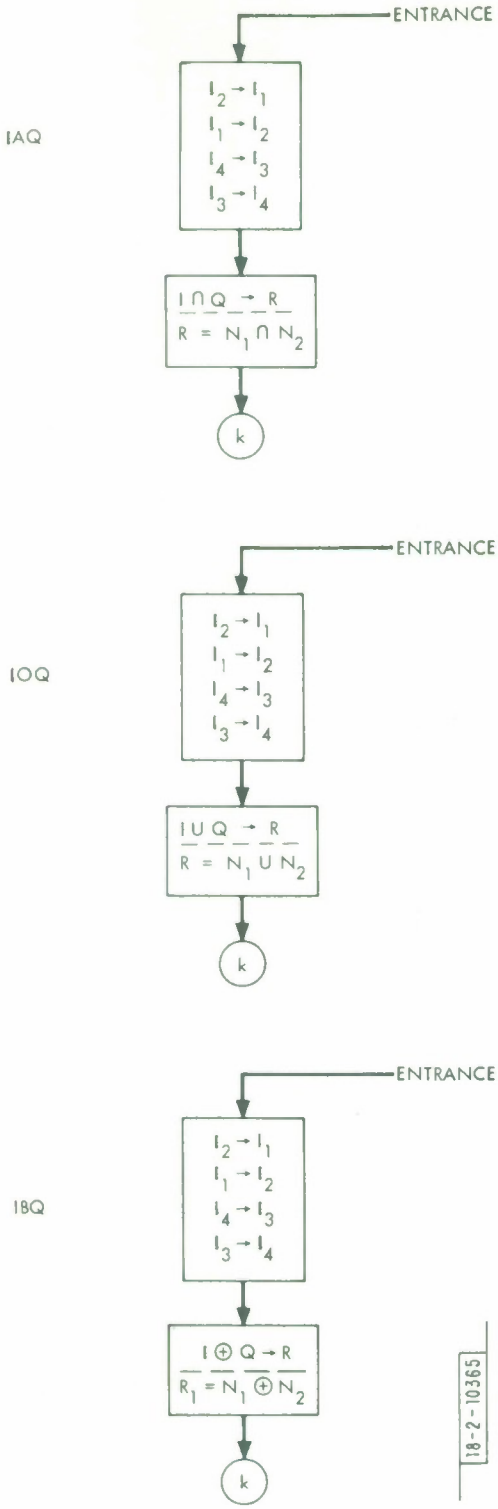


Fig. 13e. IAQ, IOQ and IBQ logic.

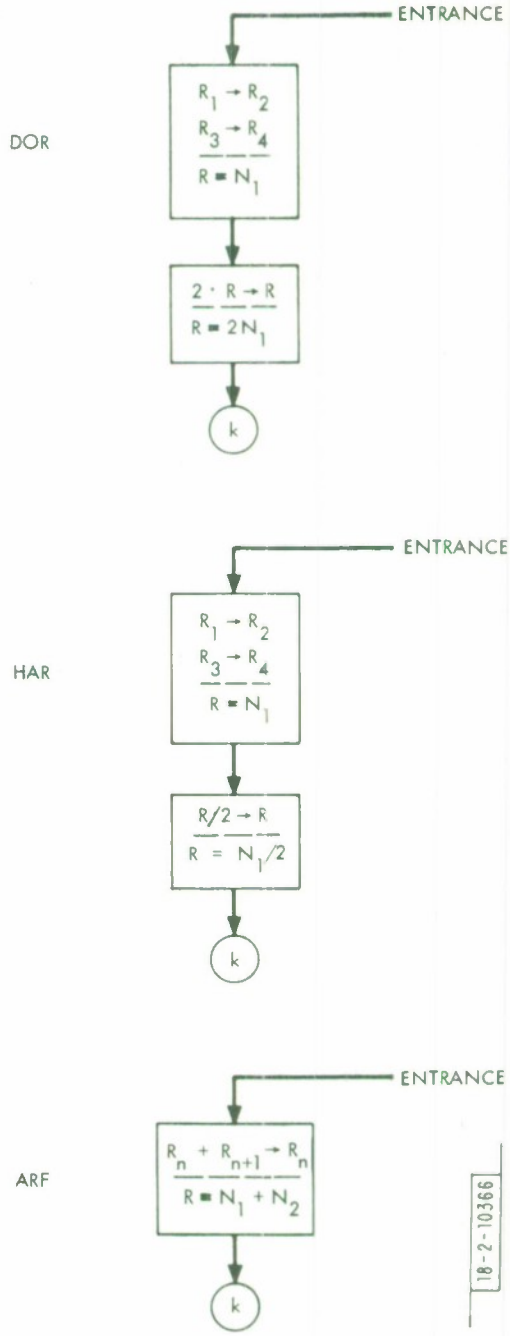


Fig. 13f. DOR, HAR and ARF logic.

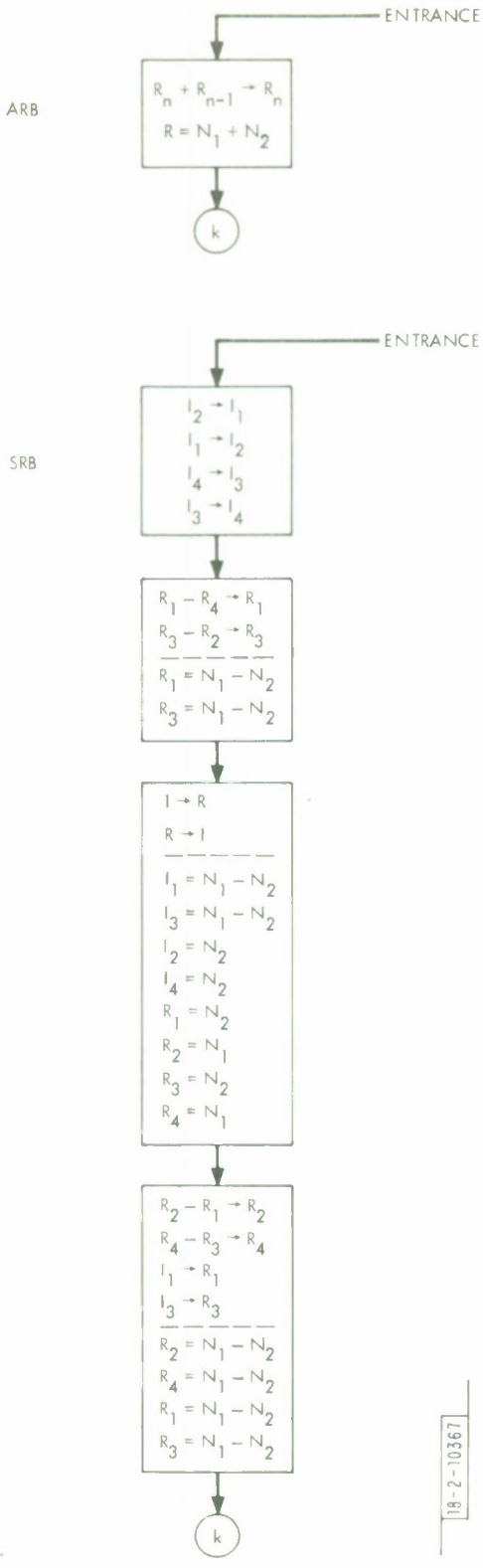


Fig. 13g. ARB and SRB logic.

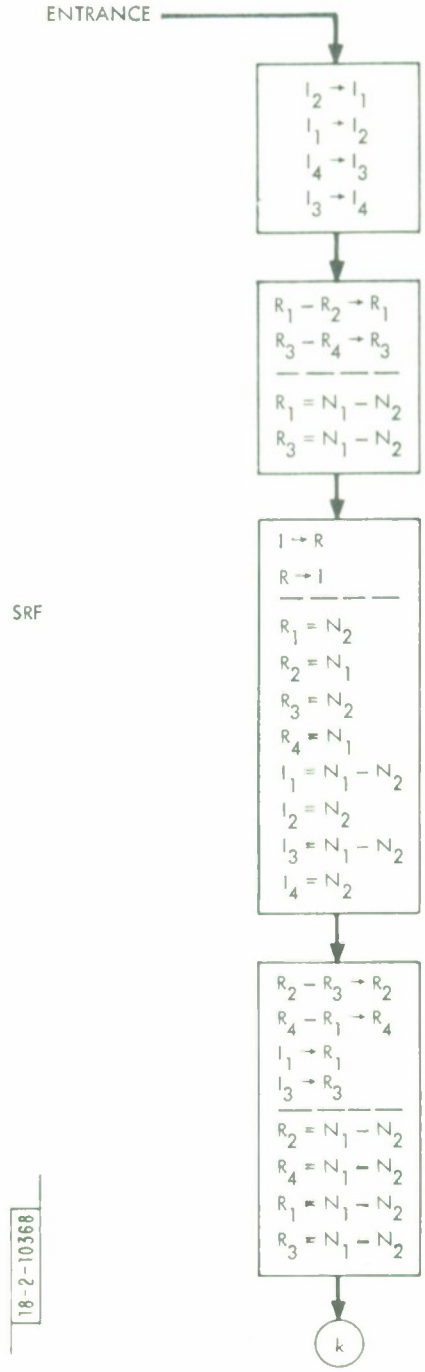


Fig. 13h. SRF logic.

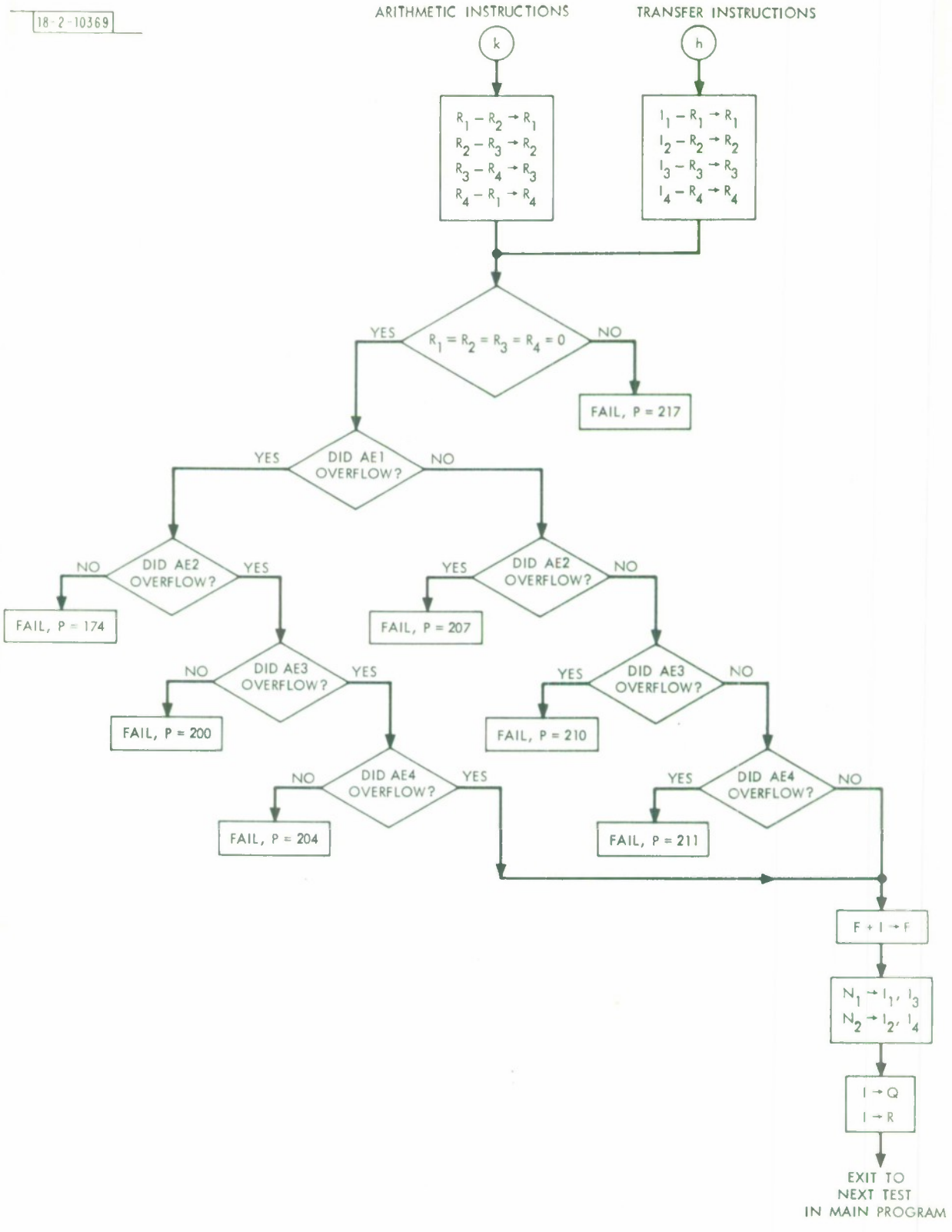


Fig. 13i. Check logic.

all instruction test programs are given in Table III (Appendix).

When both the left- and right-half arithmetic instructions have been tested, the program checks to see if all 2^{18}-1 random numbers have been tested in the arithmetic unit. If so, the arithmetic unit hardware test is complete. Otherwise, two new random numbers are generated and all the instructions are tested again.

AE Multiplier and Multiplier Overflow Test

The MUL/JOV test (Fig. 14) assures the operability of the four FDP multipliers and the logic that calculates overflow conditions associated with the product transfers.

The test works as follows:

Two random numbers are generated and multiplied together in all four multipliers in parallel. Bits 1-18 of each of the products are retrieved and compared against each other. If they are in agreement, a similar comparison is made on bits 19-36 of the products. Bits relating to TDR and TFR transfers are also retrieved. Each time a product is obtained, an overflow test is made. If one AE overflows, all ought to overflow on that particular transfer, and this fact is checked.

If there are any disparities in the product element comparisons or the overflow indications, the program stops. If not, control returns to bootstrap after 2^{18}-1 iterations. The stop conditions are:

P = 45: Failure in multiply, bits 1-18

P = 47: Failure in multiply, bits 19-36

AE1 product in $M_A(20)$

AE2 product in $M_B(20)$

AE3 product in $M_A(21)$

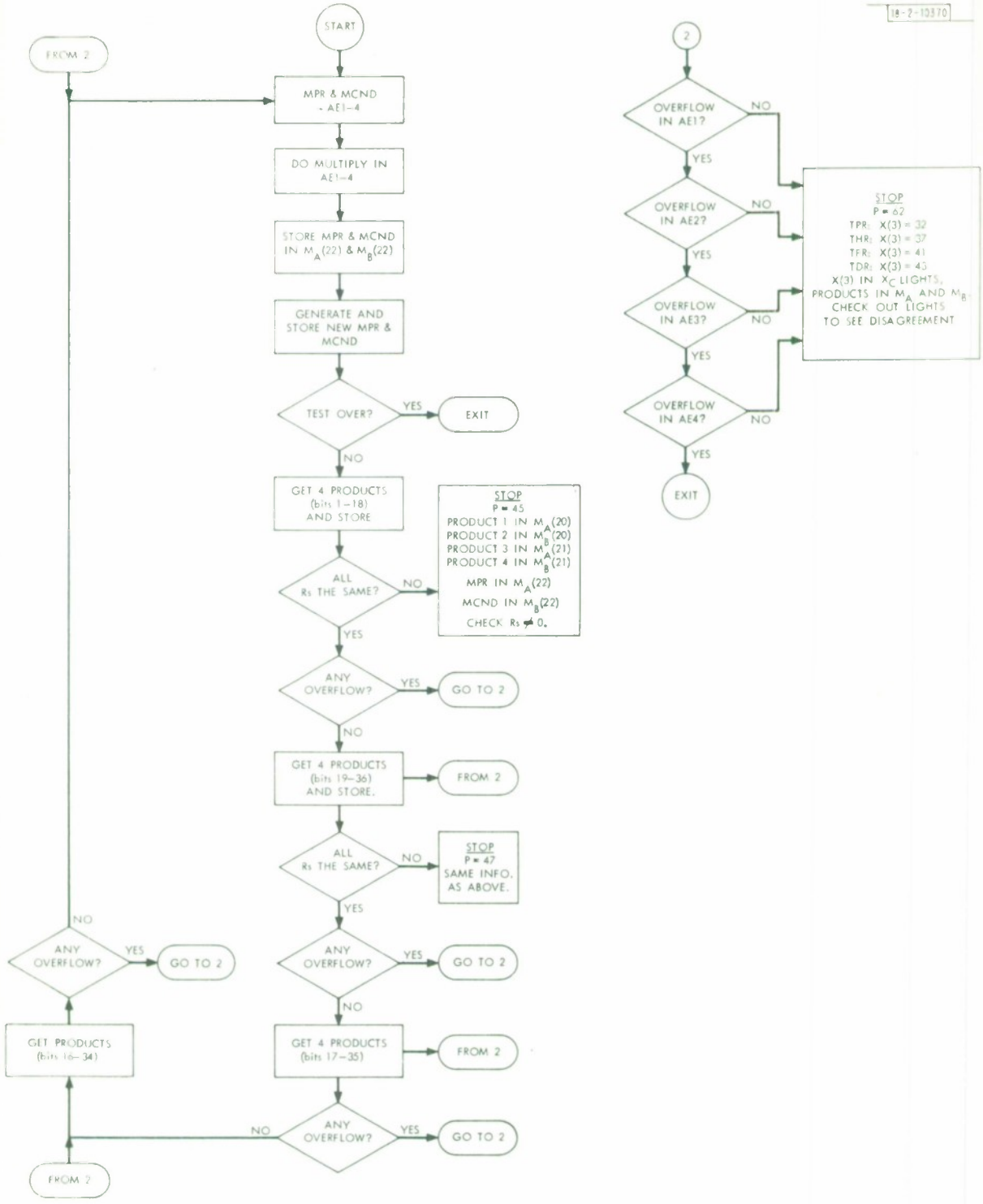


Fig. 14. Multiply/overflow test.

AE4 product in $M_B(21)$

Multiplier in M_A (22)

Multiplicand in M_B (22)

Check non-zero R registers to see where error occurred.

R_1, $R_4 \neq 0$: AE4 failed

R_2, $R_1 \neq 0$: AE1 failed

R_3, $R_2 \neq 0$: AE2 failed

R_4, $R_3 \neq 0$: AE3 failed

P = 62: Failure in overflow logic.

X(3) = 32: TPR transfer

X(3) = 37: THR transfer

X(3) = 41: TFR transfer

X(3) = 43: TDR transfer

X(3) is displayed in the X_c lights.

Check OVF lights on four AEs. The one that disagrees designates the offending AE.

Program Memory/Block Transfer Test

Creation of a diagnostic test for the 256 x 36 bit program memory presents some interesting problems unique to the FDP. In the FDP context, M_c is a "read-mostly" memory containing only program instructions. Data can be deposited in it dynamically only under program control. Once in M_c, the material becomes executable code. Information can be reaccessed only in the form of machine code. The validity of the information can be inferred only by the instructions it represents.

For example, a 36-bit word, x, is written into M_C. To establish that x was loaded properly, x must be known to correspond to a double-length FDP instruction, y. x can be inferred to have been loaded properly, if when executed, the FDP actually performed instruction y. In theory, every M_C location should be exposed to a series of bit patterns that correspond to valid FDP instructions. The series of bit patterns should be selected so that each bit in M_C is at some point required to be a 1, and at some other point a 0. In practice, this is impossible.

The approach taken involved design of a program whose sole purpose was to relocate itself in program memory. On any given iteration, the program "pushes" itself down in core one location from its previous position. The program effects this by altering appropriate locations in an image that is stored in M_A and M_B. The program then overlays itself with the modified image via the block transfer instruction. Thus the block instruction too, must function properly.

The net effect is that <u>every</u> M_C location, at some point is required to contain every instruction in this program. The set of instructions that comprise the program correspond to a series of bit patterns that fulfills the stated criterion. However, if a given bit fails, it is all but impossible to predict what the program will do. It seems reasonable to assume, however, that the program will become confused and will cease to relocate itself properly.

The test proceeds in two stages. First, the program pushes itself down from $M_C(0)$. As it moves, it fills all locations behind itself with HLT instructions. Presumably, if a fault occurs, the program will behave in one of two ways:

1. Hang up in some obscure loop.
2. Start racing wildly through core.

In the latter case, the HLT instruction will trap the program and stoppage will occur. In the former case, the user will notice that control never passes to the next test program. Manual intervention is then necessary.

If all performs well in $M_c(0)$, stage 2 begins. Here the program modifies itself to operate in bank 1 and the process begins anew in $M_c(1)$ starting at P=40. This is done to avoid clobbering the bootstrap that resides in the random access locations below 40. Otherwise, the test proceeds as it did in bank 0.

In any instance, easy interpretation of the failure mode and quick fault isolation is not possible. Careful manual examination of the two M_c banks is necessary to deduce the nature of the fault. The position of the diagnostic in the program memory stack at the time of failure should be determined and then the component instructions can be inspected. Some are not altered from iteration to iteration. The ones that require relocation (such as branches) should be evaluated in the light of the present position of the program in core. It is also possible that data memories can cause failures if the resident image of the diagnostic is altered in some illegal fashion.

In general, proper operation of the diagnostic guarantees the fidelity of the program memories to a reasonably high degree. But failure of the diagnostic does not quickly pinpoint faults. Succinctly stated, if the diagnostic fails, the FDP has a problem, hopefully in the program memories. If it does not fail, the memories are not necessarily perfect.

Ampex Core Memory Diagnostic

The Ampex core memory diagnostic residing on drum track 150 and associated with COUNT1 = 11, establishes the working status of the core memory as well as the associated controller and interface logic (LMP) when operated in the

"general" mode.

The test consists of two basic phases: (1) Exercise the memory in the so-called full-cycle mode, and (2) test the split-cycle mode. The program stresses the timing requirements of the memory system in as stringent a fashion as possible. Basically, this involves a pipelined type of test procedure wherein memory accesses are always occurring in parallel with some FDP processing.

The full-cycle phase consists of writing a full list of random numbers into M_L (the core memory). The list is then read back and checked against a locally generated version of the list (in an AE). While any given entry is being checked, an access cycle to obtain the next entry is set in motion. Thus the memory is read as rapidly as possible. When the entire list has been checked, a new list is written into M_L. The writing operation is also pipelined to push the memory as hard as possible. If a fault is detected, the FDP stops.

The split-cycle test also involves writing a full list of random numbers into M_L. However, as each location is accessed for a verification check, a new random number is written over it in accordance with split-cycle operating conventions. The entire process is pipelined to stress memory timing, i.e., while a given entry is being processed, a new datum is being written in its place, and then a new access is started to acquire the next entry. This process proceeds in a continuous loop, round and round in the memory. If an error is detected, the FDP stops.

Clearly, unless some control is exercised, both phases of the test would be open-ended. To avoid this, the program has an internal counter that limits the amount of time spent in each phase, and the amount of time spent in the diagnostic as a whole. The program starts in the full-cycle mode and does 16_{10} passes through the memory.

Then the split cycle phase is initiated and 16_{10} further iterations through the memory are performed. If no faults are found, an exit occurs.

The fault conditions are:

P = 122: Fault in M_L test

X(4) = 62: Error in full cycle mode

X(4) = 117: Error in split-cycle mode

F = Faulty address

E_0 = Bad datum

R_2 = Correct datum

R_3 = Counter.

Data Memory Diagnostic

The data memory diagnostic evaluates the two 1024 x 18 bit data memories (M_A. M_B) at as high a rate as possible. Since the worst case data and address sequencing patterns are not know for semiconductor memories, a random data approach was taken. Both memories are written rapidly in parallel with a list of randomly generated data. The list is then read back at a very high rate and checked in the AEs against a locally generated copy of the list. Any disparities constitute an error and are reported as such. The random number generator has 2^{18} -1 legal states and 1023 locations of each memory are checked. $M_A(0)$ and $M_B(0)$ are hard-wired constants. Calculations show that some 30 x 10^6 passes through the test cycle are necessary before any given memory location sees the same piece of data twice. Thus the test seems reasonably exhaustive.

The diagnostic exists in two similar forms. The first of these is the version stored on drum track 151 and corresponds to COUNT1 = 12. In this case, the program does 2048 passes through the memories before exiting. This is certainly not exhaustive, but requires about 10 seconds of running time and serves as a reasonable quick indication of the memory status. If an error is detected, the FDP stops and the console display is interpreted as follows:

P = 61: Failure in M_A

P = 100: Failure in M_B

F: Faulty location

R_3, R_4: Correct datum

M_A (F), M_B(F): Bad datum.

The second version is stored as an integral part of the Univac 1219 drum test program and is summoned whenever the drum test is. In this case, the test runs in an open-ended fashion. It was intended for use in long-term memory status checks such as overnight runs. Errors are automatically reported to the 1219 and typed on-line for quick interpretation. A sample error message is:

```
> ERROR
MB          (Faulty memory)
1047        (Faulty location)
605737      (Bad datum)
604337      (Correct datum)
```

The message states: an error was detected in bits 9 and 10 of M_B location 1047.

When an error is detected, the diagnostic program attempts to restart itself. If the memory failure is not a transient phenomenon, but rather a bona fide component failure, the test will fail again. The system makes 10_{10} attempts to restart itself before giving up. After the 10th try, the Univac ignores the FDP

and the memory diagnostic hangs up in an output loop.

It is possible to interrupt the drum test at any time and leave the memory diagnostic running in the FDP. If a failure is detected in this case, the FDP will hang up in an output loop. After the FDP is stopped, the fault condition can be ascertained just as it was in the short form version.

APPENDIX

Tabulated in this Appendix are various fault conditions that the diagnostic system might produce.

1. The contents of COUNT1 (in Univac 1219), the P register, and index register 1 (X1) are mapped into a particular fault indication.
2. Additional information to interpret the FDP console display is provided where applicable.
3. The transfer and arithmetic group test (COUNT1 = 6) is amplified in Tables II and III.

The open-ended data memory test has not been included as it types obvious error messages automatically via the 1219 teleprinter.

TABLE I

SHORT KEY TO DIAGNOSTICS (EXCEPT AE CONTROL)

COUNT1	P	X(1)	Indicated Error	Relevant Data
1	10	-	Index Memory Test--X(N) did not clear	Bits 1-4 of $M_c(7)$ point at bad location.
1	13	-	Index Memory Test--X(N) did not set	Bring X(N) into lights to see faulty bits.
2			Failure in jump test	y bits of $M_c(31)$ show where jump was to go, y bits of $M_c(36)$ show where return jump should have been. Note point of stoppage and whether XJP was in correct M_c location.
3	21	-	Index Jump Test--JPX failed	X_c lights show index count, R_1 shows AE count
3	24	-	Index Jump Test--JNX failed	X_c lights show index count, R_2 shows AE count
3	146	(see Fig. 5)	Failure in AE jump test	Compare Fig. 5 and X_c lights for actual error
3	224	175	Index AR.Test--Failure to form N + 1	R_1 =N, X(17) = N+1, R_2 = difference between results.
3	224	207	Index AR.Test--Failure to form N - 1	R_1 = N, X(16) = N - 1, R_2 = difference between results.
3	224	221	Index AR.Test--failure to form -1-N	R_1 = N, X(15) = -1-N, R_2 = difference between results.
3	303	255	Index Skip Test--claimed X = 0, AE disagreed	In all cases, X_A and X_B lights display X(1), the stop point. The X_c lights display N, the random number causing the problem. See Fig. 7.
3	303	264	Index Skip Test--claimed X = 0 after decision $X \neq 0$	
3	303	270	Index Skip Test--claimed $X \leq -1$ after decision $1 > 0$.	
3	303	273	Index Skip Test--claimed $X < 0$ after decision $x > 0$	
3	303	275	Index Skip Test--claimed $X \geq 0$ after decision $X < 0$	
3	303	277	Index Skip Test--claimed $X \geq 1$ after decision $X < 0$	
3	303	302	Index Skip Test--claimed $X > 0$ after decision $X < 0$	

TABLE 1 (Continued)

4	6		SKM Test-failed to unconditionally skip	0	To establish N, look at bits 1-4 of $M_c(14)$. If failure is on a first instruction, flag is probably bad. If on subsequent instruction, problem is probably in delay line.
4	10		SKM Test-failed to skip first of next two instructions	0	
4	11		SKM Test-failed to skip second of next two instructions	0	
4	13		SKM Test-failed to skip first of next three instructions	1	
4	14		SKM Test-failed to skip second of next three instructions	1	
4	15		SKM Test-failed to skip third of next three instructions	1	
4	17		SKM Test-failed to skip first of next four instructions	Flag (N) = 0	
4	20		SKM Test--failed to skip second of next four instructions		
4	21		SKM Test-failed to skip third of next four instructions		
4	22		SKM Test--failed to skip fourth of next four instructions		
5	103	6	FGP Test--F X path cannot send 0s		Faulty bits should be apparent in F lights
5	103	13	FGP Test--F X path cannot send 1s		Faulty bits should be apparent in F lights
5	103	27	FGP Test--Lost bit In left shift of 18 places		Inspect the F lights
5	103	40	FGP Test--Lost bit in right shift of 18 places		Inspect the F lights
5	103	73	FGP Test--F adder failed to compute N_1 and N_2		R_1 and R_2 contain $N_1 + N_2$. F contains result as computed by F adder. R_4 Contains difference between this and the AEs result.
5	103	101	FGP Test--F adder failed to compute $N_1 - N_2$		
5	143		BRA Test-failure to calculate $N_1 + N_2$		$R_3 = N_1, R_4 = N_2, F$ = Result. R_1 = difference
5	324	162	Memory Reference--Memory↔AE paths did not send 0s		All 1s and Rs should be 0. See which is not
5	324	171	Memory Reference--Memory↔AE paths did not send 1s		All 1s = -1, all Rs = 0. Any I ≠ -1 is bad
5	324	177	Memory Reference--Memory↔F paths did not send 0s		All Is, Rs and F should = 0. Check if so
5	324	210	Memory Reference--Memory↔F paths did not send 1s		All 1s, F = -1, all Rs = 0. Check if so
5	324	223	Memory Reference--Failure of M_A or M_B transfer inhibit		If R_1 and $R_2 \neq 0$, α failed. If R_3 and $R_4 \neq 0$, β failed

TABLE 1 (Continued)

5	324	240	Memory Reference--M_D(I) location did not clear	(I) can be inferred from bits 27-30 of M_c(234) Bring M_D (I) into lights and examine
5	324	245	Memory Reference--M_D(I) location did not set	
5	326		Failure of address adder to compute N + 1	F = N + 1, N - 1. M_D(1) = N. ADR_A, ADR_B lights should agree with F. $R_1 \neq 0$: X_A adder bad. $R_2 \neq 0$: X_B adder bad.
5	330		Failure of address adder to compute N - 1	
6			See Tables 2 and 3	
7	45	-	Multiplier Failure--bits 1-18 of product	AE1, 2 product in $M_{A,B}$(20). AE3, 4 product in $M_{A,B}$(21). MPR, MCND in $M_{A,B}$(22). R_1, $R_4 \neq 0$: AE4 bad; R_2, $R_1 \neq 0$: AE1 bad; R_2, $R_3 \neq 0$: AE2 bad; R_4, $R_3 \neq 0$: AE3 bad.
7	47		Multiplier Failure--bits 19-36 of product	
7	62	32	JOV failure on TPR	X_c lights display X(3) which is actual stop code. Check OVF lights in AE. One in disagreement tags bad AE.
7	62	37	JOV failure on THR	
7	62	41	JOV failure on TFR	
7	62	43	JOV failure on TDR	
10	-	-	Failure in program memory diagnostic	See text. No simple interpretation.
11	116	-	M_L fault, full-cycle mode	R_3 = counter, F = address, E_0 = bad datum, R_2 = correct datum.
11	117	-	M_L fault, split-cycle mode	
12	61		M_A fault	F = address; R_3, R_4 = correct datum; M_A (F), M_B(F) = faulty datum
12	100		M_B fault	

TABLE II

AE ARITHMETIC/TRANSFER TEST--

OVERFLOW FAILURE CONDITIONS

P	AE1 Overflow FF	AE2 Overflow FF	AE3 Overflow FF	AE4 Overflow FF
174	1	0	not checked	not checked
200	1	1	0	not checked
204	1	1	1	0
207	0	1	not checked	not checked
210	0	0	1	not checked
211	0	0	0	1

TABLE III

TRANSFER AND ARITHMETIC INSTRUCTIONS

F Register (Base 8)	Failure[1, 2]	Legal Overflow Possible	Routine[3] Start P	End P
1	Right half transfer	No	15	25
2	Left half transfer	No	27	36
3	Left half COR	No	40	42
4	Left half MAR	Yes	44	46
5	Left half CSR	Yes	50	52
6	Left half IMR	Yes	54	56
7	Left half RMI	Yes	60	62
10	Left half IPR	Yes	64	66
11	Left half IAQ	No	70	72
12	Left half IOQ	No	74	76
13	Left half IBQ	No	100	102
14	Left half DOR	Yes	104	106
15	Left half HAR	No	110	112
16	Left half ARF	Yes	114	116
17	Left half ARB	Yes	120	122
20	Right half SRB	Yes	124	133
21	Right half SRF	Yes	135	144
22	Right half COR	No	40	42
23	Right half MAR	Yes	44	46

TABLE III (continued)

24	Right half CSR	Yes	50	52
25	Right half IMR	Yes	54	56
26	Right half RMI	Yes	60	62
27	Right half IPR	Yes	64	66
30	Right half IAQ	No	70	72
31	Right half IOQ	No	74	76
32	Right half IBQ	No	100	102
33	Right half DOR	Yes	104	106
34	Right half HAR	No	110	112
35	Right half ARF	Yes	114	116
36	Right half ARB	Yes	120	122
37	Right half SRB	Yes	124	133
40	Right half SRF	Yes	135	144

---

1 The last step in the test of an instruction or class of instructions is a subtraction that should produce 0s in R_1, R_2, R_3 and R_4. If one or more Rs is non-zero, there is an instruction failure and the machine will stop with $P = 217_8$.

2 An instruction also fails if all four overflow flip-flops are not identical. The P address where the machine stops and the failure conditions are listed in Table II.

3 After observing F and P, the error can be pinpointed by stepping the machine

TABLE III (continued)

through the routine that failed. First, master clear, restart the program at P = 15, run the program until it reaches the start address of the routine that failed, and then step the machine through the program observing closely the contents of all AE registers. Ignore F.

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Lincoln Laboratory, M.I.T.	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP None

3. REPORT TITLE

FDP Diagnostic System

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Technical Note

5. AUTHOR(S) *(Last name, first name, initial)*

Blankenship, Peter E.
McHugh, Paul G.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
22 February 1972	64	None

8a. CONTRACT OR GRANT NO. F19628-70-C-0230 b. PROJECT NO. 649L c. ARPA Order 1559 d.	9a. ORIGINATOR'S REPORT NUMBER(S) Technical Note 1972-15 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* ESD-TR-72-58

10. AVAILABILITY/LIMITATION NOTICES

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Air Force Systems Command, USAF Advanced Research Projects Agency, Department of Defense

13. ABSTRACT

A diagnostic system consisting of a series of programs to check the operation of each subset of the computer hardware is described as are the details of each diagnostic test. All fault conditions and the concomitant console display interpretation for each condition are tabulated for easy deciphering.

14. KEY WORDS

Fast Digital Processor (FDP)
diagnostic system